UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

Pruebas de mutación para Ruby

Iván Díaz Moreno
Tutor: Esther Guerra Sánchez

Julio 2020

Pruebas de mutación para Ruby

AUTOR: Iván Díaz Moreno
TUTOR: Esther Guerra Sánchez

Escuela Politécnica Superior
Universidad Autónoma de Madrid
Julio 2020

Resumen

Una de las prácticas más recomendables y usadas por los programadores es la realización de casos de prueba para comprobar la calidad, robustez y fiabilidad del código programado. En ocasiones, la inexperiencia del programador o la falta de tiempo puede provocar que la efectividad de estos casos de prueba no sea la mejor posible.

Una forma de comprobar la efectividad y mejorar la calidad de los casos de prueba programados es mediante el uso de pruebas de mutación. Estas consisten en realizar en el código pequeñas modificaciones, llamadas mutaciones, que den como resultado nuevos códigos con fallos que un programador podría haber cometido. Estos nuevos códigos generados son los llamados mutantes. El objetivo es ejecutar los casos de prueba sobre estos mutantes y comprobar si los fallos provocados por estos cambios son detectados. En caso de no ser así, el programador podrá realizar nuevos casos para gestionar estos nuevos mutantes creados y mejorar así su cobertura de casos de prueba.

Con el fin de ayudar en la mejora de la calidad del código y su eficacia, este Trabajo Fin de Grado tiene como objetivo desarrollar una aplicación que permita generar de forma automática mutaciones para un programa en lenguaje Ruby y que evalúe la efectividad de los casos de prueba de este programa sobre los mutantes creados.

Con esta aplicación el programador podrá conocer de forma sencilla que mutantes se han generado y podrá obtener un informe sobre los resultados obtenidos que le permita conocer la eficacia de sus casos de prueba.

Debido a que las posibilidades en el mundo de la programación son, aunque finitas, muy variadas, las necesidades del programador también pueden serlas. Por este motivo, esta aplicación también busca proporcionar al usuario la posibilidad de generar sus propias opciones de mutación y su gestión.

Palabras clave

Pruebas de mutación, operador de mutación, mutante, árbol de sintaxis abstracta (AST), mutación, casos de prueba, tests.

Abstract

One of the most recommended practices used by programmers is to carry out test cases to check the quality, robustness and reliability of the programmed code. Sometimes, the inexperience of the programmer or lack of time can cause that the effectiveness of these test cases are not the best possible.

One way to check the effectiveness and improve the quality of scheduled test cases is by using mutation testing. These consist of making small modifications in the code, called mutations, that result in new codes with failures that a programmer could have committed. These new generated codes are called mutants. The objective is to run the test cases on these mutants and check whether the failures caused by these changes are detected. If this is not the case, the developer can add new cases to manage these newly created mutants and thus improve their test case coverage.

To help improving the quality of the code and its effectiveness, this Final Degree Project aims to develop an application that automatically generates mutations for a program written in the Ruby programming language and then evaluates the effectiveness of the test cases of this program on the mutants created.

With this application the programmer will be able to know in a simple way which mutants have been generated and will be able to obtain a report on the obtained results that allows knowing the efficacy of the test cases.

Because the possibilities in the world of programming are, although finite, very varied, so are the programmers needs. For this reason, this application also seeks to provide the user with the possibility of generating their own mutation options and their management.

Keywords

Mutation, mutation operator, mutant, abstract syntax tree (AST), mutation, test cases, tests.

Agradecimientos

A mis padres por darme esta oportunidad y haberme apoyado en todo momento, sin ellos este grado no habría sido posible.

A mis amigos de siempre, por estar en esos momentos en los que más los necesitaba para desconectar del trabajo del día a día.

A mis compañeros y amigos de la universidad por el apoyo mutuo y buenos momentos que hemos vivido en estos años.

A mi tutora Esther por tutelar este trabajo fin de carrera y la ayuda que me ha prestado.

A todos aquellos que han formado parte de mi vida y que me han hecho ser la persona que soy hoy.

Gracias a todos.

INDICE DE CONTENIDOS

INDICE DE FIGURAS

INDICE DE TABLAS

1 Introducción

Puesto que en un principio la aplicación solo permitirá mutar programas desarrollados en Ruby, esta misma aplicación ha sido desarrollada en este mismo lenguaje. Por lo que a lo largo de este documento siempre que haya referencias a códigos o estructuras de programación, se estará hablando de Ruby a no ser que se especifique lo contrario.

1.1 Motivación

Ruby [12] es un lenguaje de programación orientado a objetos fuertemente influenciado por otros lenguajes como Python o Perl. En la actualidad es un lenguaje que goza cada vez de más fama entre los desarrolladores de código que buscan un lenguaje sencillo para programar y trabajar. Además de tratarse de un lenguaje orientado a objetos, se trata de un lenguaje interpretado, lo cual significa que no necesita de un compilador que lo traduzca a lenguaje máquina antes de la ejecución. En su lugar el propio interprete se encarga de compilarlo y ejecutarlo al mismo tiempo. Como consecuencia, se trata de un lenguaje dinámico, como lo es Python, que permite una programación más libre y sencilla, además de la posibilidad de ser modificado durante el tiempo de ejecución.

Al ser un lenguaje dinámico, el código programado puede contener errores que pasen inadvertidos en tiempo de compilación y sólo sean descubiertos en tiempo de ejecución. Por eso en este lenguaje es especialmente importante complementar el código con pruebas unitarias que ejerciten suficientemente el código y que sean de la mejor calidad posible.

Por este motivo, Ruby es uno de los lenguajes más usados en la actualidad y el elegido para desarrollar la aplicación tratada en este Trabajo Fin de Grado. En el último año ha pasado de ocupar la posición 18 de los más usados a la 11 [13].

Entre las distintas fases que nos encontramos durante la creación de alguna aplicación o programa tanto en Ruby como en cualquier lenguaje, la fase de pruebas es una de las más importantes de todas. Toda aplicación debe tener un buen conjunto de casos de prueba que lo respalde para dar tranquilidad a sus desarrolladores y asegurar la calidad y correcta funcionalidad de este. La importancia de estas pruebas es tal, que en la actualidad existen un gran número de herramientas y programas que ayudan a comprobar un código programado en todos sus aspectos. Entre todas ellas nos encontramos con las pruebas de mutación [14], estas consisten en realizar en el código pequeñas modificaciones, llamadas mutaciones, que den como resultado nuevos códigos con fallos que un programador podría haber cometido. Estos nuevos códigos generados son los llamados mutantes. El objetivo es ejecutar las pruebas unitarias, programadas para ese código, sobre estos mutantes y comprobar si los fallos provocados por estos cambios son detectados, por lo que su objetivo es medir la calidad de las pruebas programadas basándose en la hipótesis de que si un conjunto de pruebas es capaz de distinguir un programa de sus mutantes, entonces probablemente también sea bueno en la detección de errores.

Aunque en la actualidad existen muchas opciones para asegurar la correcta funcionalidad de nuestro código y casos de prueba como ya se ha mencionado, muchas de estas herramientas no detectan siempre ciertos casos que las pruebas de mutación si son capaces de detectar. En muchas ocasiones, los mutantes son capaces de mostrar errores no descubiertos por el programador y sus casos de prueba. Mientras que otras herramientas o no los detectan o los dan por bien cubiertos.

Las pruebas de mutación no son muy usadas por la complejidad de realizar esta tarea de forma manual y su alto coste de tiempo en ser desarrolladas. Por este motivo se ha pensado en desarrollar una aplicación que se encargue de realizar todo el proceso automáticamente y que permita al usuario obtener unos resultados completos y concisos. Ahorrando a este una gran cantidad de tiempo e incentivándolo a hacer uso de las pruebas de mutación. Esta es una práctica muy recomendada que permitirá a los desarrolladores mejorar la cobertura de sus pruebas.

Finalmente, aunque existen herramientas para aplicar pruebas de mutación a Ruby, estas herramientas no son extensibles, no proporcionan una GUI que facilite su uso, sino que la utilización es por terminal mediante el uso de comandos, y no proporcionan mecanismos usables de visualización de resultados, por lo que su uso no es intuitivo ni sencillo para el usuario.

1.2 Objetivos

El objetivo es por tanto desarrollar una aplicación para Ruby que permita la generación automática de mutaciones y la obtención de los resultados de aplicar los casos de prueba de dichos códigos sobre los mutantes generados. Permitiendo al usuario tener una herramienta efectiva para poder medir la calidad de sus pruebas.

Para la creación de los mutantes el objetivo es poder hacer uso del AST del código de entrada. El AST, árbol de sintaxis abstracto, que es una representación en forma de árbol de la estructura sintáctica simplificada del código fuente, que permite el fácil recorrido de esta y su modificación.

La aplicación desarrollada busca ser lo más sencilla y clara posible, con el objetivo de incentivar su uso y dotar a los desarrolladores de una herramienta que permita mejorar su cobertura de pruebas mediante el uso de pruebas de mutación.

Con este objetivo, la aplicación contará con cuatro funcionalidades básicas: **a)** poder generar mutantes a partir de un código Ruby dado, **b)** poder evaluar los resultados de ejecutar los casos de prueba sobre los mutantes generados, **c)** permitir al usuario configurar sus propias operaciones de mutación para casos específicos que puedan requerir su código y, **d)** poder eliminar aquellas operaciones de mutación creadas anteriormente que ya no se desean usar.

Con el fin de que la aplicación sea más sencilla e intuitiva de usar, el objetivo es desarrollar una pequeña GUI, que permita al usuario navegar fácilmente entre las distintas funcionalidades y hacer uso de estas, además de proporcionar un espacio en el que mostrar los resultados de forma clara y ordenada.

Por este motivo la aplicación podría dividirse en dos partes. Por un lado, un API que contiene toda la funcionalidad necesaria para implementar todas las operaciones requeridas por la aplicación que ya se han mencionado anteriormente (generar mutantes, analizar resultados, generar opciones de mutación y gestionar dichas opciones) y por otro lado un GUI que permite hacer uso de todas las funcionalidades que provea el API de una forma mucho más sencilla por parte del usuario.

Finalmente, la aplicación contará con dos opciones de ejecución para que elijan la que mejor se adapte a sus necesidades. Podrán escoger entre un modo con gestión de usuarios y otro sin él. De esta forma si escogen el modo con gestión de usuarios la aplicación permitirá tener

varios usuarios registrados y las acciones de un usuario no afectaran a las acciones realizadas por otro en la creación y eliminación de operadores, mientras que, si escogen el modo sin gestión de usuario, cualquier usuario que acceda al dispositivo con este modo, podrá visualizar los operadores creados por otros usuarios y si lo desea podrá eliminarlos libremente. Debe matizarse, que el modo con gestión de usuarios permitirá acceder a la aplicación tanto con usuarios registrados en el sistema, como en un modo de usuario sin registro que realizará las mismas funciones que el modo sin gestión de usuarios.

1.3 Organización de la memoria

La memoria consta de los siguientes capítulos:

- **Capítulo 1**: Introducción. Este capítulo presenta el contenido del TFG y algunas nociones básicas.
- **Capítulo 2**: Estado del arte. Se presenta el concepto de pruebas de mutación junto con un análisis de las herramientas ya existentes y sus puntos más fuertes y débiles. Seguidamente se explica la motivación que ha llevado a realizar este proyecto a partir del análisis realizado y finalmente se describen las herramientas más importantes usadas para llevarlo a cabo.
- **Capítulo 3**: Diseño. Se exponen los aspectos más relevantes del diseño de la aplicación y las decisiones tomadas antes de comenzar con el desarrollo de esta.
- **Capítulo 4**: Desarrollo. Se expone el proceso seguido para desarrollar la aplicación y detalles relevantes de la implementación de esta.
- **Capítulo 5**: Pruebas y resultados. Este capítulo expone las pruebas realizadas con la aplicación para comprobar su correcto funcionamiento y expone los resultados obtenidos tras la realización del proyecto.
- **Capítulo 6**: Conclusiones y trabajo futuro. Finalmente, en el último capítulo se muestran las conclusiones obtenidas tras la realización de este Trabajo Fin de Grado y los posibles trabajos futuros que se pueden realizar sobre la aplicación resultante.

2 Estado del arte

En este apartado abordaré el concepto de pruebas de mutación, expondré algunas herramientas ya existentes que permiten realizar este tipo de pruebas, evaluaré las limitaciones encontradas en estas y finalmente hablaré sobre las tecnologías usadas para llevar a cabo este proyecto.

2.1 Pruebas de mutación

Entre las distintas fases por las que pasa un software a lo largo de su ciclo de vida, una de ellas es la fase de pruebas. Esta es una de las fases más importantes debido a la gran labor que realizan en la detección de errores y la búsqueda de la mayor robustez y calidad del programa desarrollado.

Durante esta fase los desarrolladores implementan un conjunto de pruebas con el objetivo de detectar comportamientos no deseados en el código programado y comprobar la calidad de este. En general las pruebas que suelen implementarse se pueden agrupar en dos: las pruebas de caja blanca, que tienen como objetivo comprobar el correcto funcionamiento interno de todos los elementos del programa conociendo la estructura interna de los mismos y, las de caja negra, que buscan comprobar el correcto funcionamiento en base a las entradas y salidas de los distintos componentes del programa sin conocer la composición interna de cada uno. Ambas podrían ejemplificarse en la vida real con la compra de una furgoneta. Las pruebas de caja negra serían las que realizaría el cliente antes de comprar el vehículo: ver que al dar al botón de encender las luces se encienden, que al pisar el acelerador el motor aumenta sus revoluciones, etc. Estas son pruebas que se realizan sin conocer el funcionamiento interno y se evalúan en función a una respuesta esperada provocada por una acción. Y las pruebas de caja blanca serían las que realizaría el mecánico para comprobar que el vehículo es completamente seguro y funcional para ser vendido: comprobar todos los mecanismos internos, uno por uno, para asegurarse de su correcto funcionamiento y estado.

Las pruebas de mutación [14] se tratan de un tipo de pruebas de caja blanca que tienen como objetivo medir la calidad del conjunto de casos de prueba de un programa. Estas consisten en realizar pequeños cambios en el código original del programa, a los que denominamos mutaciones. Para generarlas, es necesario aplicar un operador de mutación al código que se desea mutar. Este operador de mutación no es más que el cambio que se desea realizar. Por ejemplo, un operador de mutación sería cambiar un signo '+' por un signo '-' o eliminar una línea concreta del código entre otros muchos. El resultado es un nuevo código, denominado mutante, contra el que se debe comprobar si los casos de prueba que se poseen son capaces de detectar los cambios. Este tipo de pruebas generalmente se realizan durante el periodo de pruebas unitarias, que es el periodo en el que se comprueba la correcta funcionalidad de cada componente del programa, de uno en uno y por separado.

Debe mencionarse, que los mutantes que se generan deben ser programas válidos del lenguaje bien formados, que luego puedan ejecutarse.

Una vez que se han generado los mutantes, se ejecutan los casos de prueba sobre ellos. Si el mutante pasa todos los casos entonces decimos que ha sobrevivido, lo que significa que los casos de prueba no son capaces de detectar el fallo introducido y por lo tanto las pruebas que realizadas no están comprobando la línea que se mutó correctamente, mientras que, si el

mutante no es capaz de pasar todos los casos de prueba, entonces decimos que el mutante ha muerto y en ese caso podemos afirmar que los casos de prueba tienen la robustez suficiente como para detectar el fallo introducido.
El objetivo de las pruebas de mutación es medir la calidad de los casos de prueba programados, que deben ser lo suficientemente efectivos como para detectar los fallos en el código mutante. La calidad del conjunto de casos de prueba viene dada por el llamado "porcentaje de mutación", que se calcula como el porcentaje de mutantes muertos sobre el número total de mutantes. Si el “porcentaje de mutación” se considera bajo, el desarrollador podría mejorar la calidad de su conjunto de casos de prueba añadiendo nuevas pruebas capaces de matar a los mutantes vivos. Por este motivo las pruebas de mutación son una muy buena opción para analizar la cobertura de nuestras pruebas (comprobar que todas las líneas de código están bien cubiertas por estas ante cambios) sobre nuestro código.

El proceso de generación de mutantes y la posterior evaluación de los casos de prueba sobre los mismos suele ser una tarea con un alto coste de tiempo y muy difícil de realizar sin tener una herramienta automatizada, además de no poder ser usada en las pruebas de caja negra por la necesidad de tener que acceder al código fuente para poder ser modificado. A pesar de esto su utilización proporciona al desarrollador grandes ventajas. En primer lugar, saca a la luz nuevos tipos de errores difícilmente detectables a simple ojo por los desarrolladores, es uno de los métodos más efectivos para detectar errores ocultos que son indetectables mediante el uso de técnicas de pruebas convencionales, ayuda a medir la cobertura de las pruebas, detectar ambigüedades y obtener sistemas con un índice más alto de estabilidad y fiabilidad.

2.2 Herramientas relacionadas

En esta sección se mostrarán algunas herramientas ya existentes para la realización de pruebas de mutación en programas desarrollados en Ruby.

2.2.1 Crude-Mutant

Es una gema de Ruby [1] que permite un tipo muy específico de mutación. Esta gema ha sido diseñada para eliminar líneas de código y comprobar cuales de ellas no son necesarias para pasar las pruebas.

Así pues, esta gema aporta la funcionalidad necesaria para dado un fichero de entrada, generar tantos mutantes como líneas tenga el fichero, eliminando en cada mutante una línea concreta y posteriormente ejecutar las pruebas sobre estos mutantes para comprobar en qué casos las pruebas se siguen pasando sin detectar errores y en cuáles no.

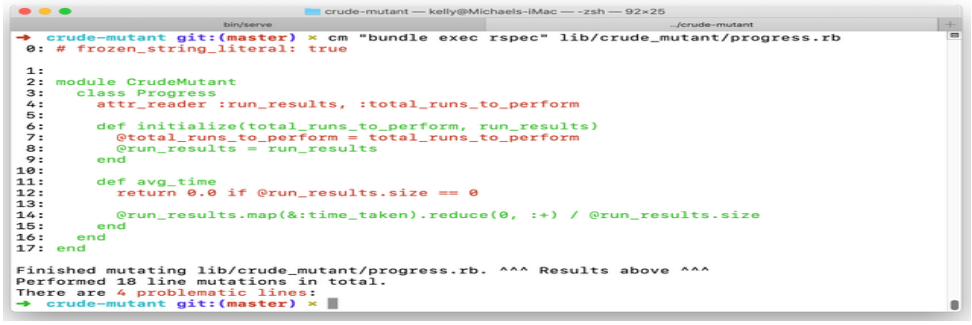


Figura 1 - Salida de ejecución de crude mutant

El objetivo principal de esta herramienta es detectar aquellas líneas que no están siendo comprobadas por los casos de prueba programados.

La **Figura 1**, muestra un ejemplo de ejecución de esta herramienta. El resultado obtenido es el fichero mutado, marcando en rojo aquellas líneas que al ser eliminadas han generado errores y, marcando en verde aquellas que al ser borradas no han sido detectadas como fallo en las pruebas.

Tras ejecutar la herramienta el usuario tiene una representación de que líneas no están siendo bien cubiertas por sus casos de prueba.

Aunque es una herramienta útil, la funcionalidad que proporciona es algo escasa, ya que las mutaciones que permite generar son únicamente de borrado de líneas y por tanto no es capaz de detectar ciertos fallos que pueden estar más ocultos.

2.2.2 Mutant

Mutant es otra gema Ruby que otorga a los usuarios una herramienta para realizar pruebas de mutación automatizadas con mayores posibilidades que la herramienta vista en el apartado 2.2.1. En este caso, Mutant permite a los usuarios realizar un amplio número de mutaciones diferentes con la limitación de no dejar elegir a los mismos los operadores que desea aplicar en cada mutación [2].

El objetivo de Mutant es el de ayudar al usuario a medir la calidad de sus pruebas.

En este caso, como ya he comentado, nos encontramos ante una herramienta que ya no solo permite realizar la mutación de eliminación de líneas, sino que también permite realizar otros tipos de mutaciones como cambiar operadores, reemplazar llamadas a funciones similares, eliminar ciertos elementos o agregar otros nuevos al código proporcionado entre otros muchos.

```
$ bundle exec mutant --use rspec -I lib/ -r team Team

# ... bunch of results ...

-----------------------
Mutant configuration:
Matcher:         #<Mutant::Matcher::Config match_expressions: [Team]>
Integration:     Mutant::Integration::Rspec
Jobs:            8
Includes:        ["lib/"]
Requires:        ["team"]
Subjects:        10
Mutations:       317
Results:         317
Kills:           295
Alive:           22
Runtime:         5.30s
Killtime:        26.00s
Overhead:        -79.62%
Mutations/s:     59.82
Coverage:        93.06%
```

Figura 2 - Salida ejecución Mutant

```
evil:Team#eql?:/Users/i848350/git/github/neontapir/managertools-logging/lib/team.rb:59:e1a55
@@ -1,6 +1,6 @@
 def eql?(other)
   if other.respond_to?(:team)
-    team == other.team
+    team
   end
 end
```

Figura 3 - Salida ejecución Mutant

La **Figura 2** nos muestra el comando necesario para ejecutar la herramienta y la salida con los datos genéricos obtenidos de realizar todas las mutaciones posibles al código proporcionado y de haber pasado las pruebas proporcionadas a cada mutante. En esta salida el usuario puede tener una visión general del proceso de mutación, conociendo el número de mutantes generados, cuantas de ellas han sobrevivido y cuantas han muerto además del porcentaje de cobertura que tiene sus pruebas.

En la **Figura 3**, se muestra un fragmento de la salida en la que el usuario puede tener una visión más específica de una de las partes de código mutadas y si ha sido detectado el cambio por las pruebas o no. Indicando en rojo aquellas partes que no se han detectado y en verde las que sí.

2.2.3 Mutest

Al igual que la herramienta anterior, Mutest se trata de una gema para Ruby que permite realizar mutaciones a nuestro código y conocer la cobertura de nuestras pruebas [3]. En este caso la herramienta permite realizar diferentes mutantes sobre un código dado o sobre un proyecto dado. Se trata de una bifurcación de la gema Mutant que elimina ciertas operaciones de mutación, agrega otras y modifica algunas de las ya existentes. Además, añade nueva funcionalidad que permite, mediante comentarios, especificar que líneas no se desea mutar.

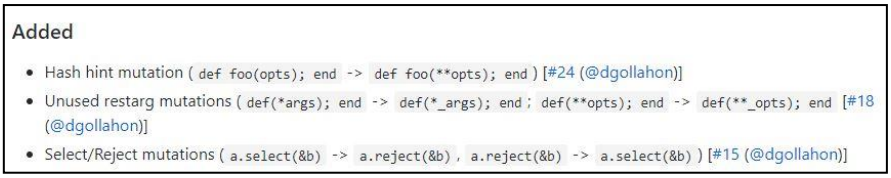


Figura 4- Ejemplo operaciones creadas por mutest

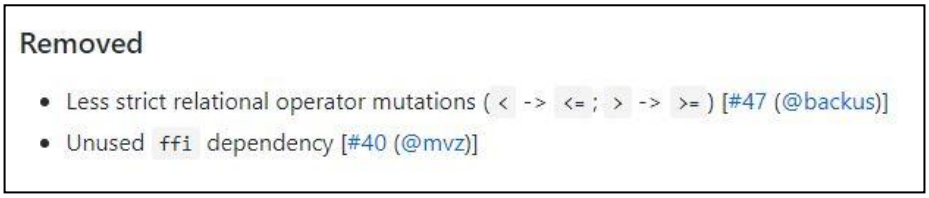


Figura 5- Ejemplo operaciones eliminadas por mutest

En las **Figuras 4 y 5** podemos observar algunos ejemplos de opciones de mutación que Mutest ha añadido o eliminado con respecto a Mutant [4].

El objetivo de esta herramienta es ayudar a detectar la cobertura de las pruebas desarrolladas y fomentar la realización de código más robusto. La diferencia con Crude-Mutant es abismal, permitiendo realizar operaciones de mutación mucho más variadas y obteniendo por lo tanto mejores resultados para medir la cobertura de los casos de prueba desarrollados. Además de permitir al usuario seleccionar que líneas del código desea que sean mutadas y cuáles no.
A continuación, se muestra la salida de la ejecución de esta herramienta en la **Figura 6** y la **Figura 7**.

Figura 6 - Salida ejecución Mutest

Figura 7- Salida ejecución Mutest

La **Figura 6** muestra los resultados de las mutaciones obtenidas desde un punto de vista más genérico, permitiendo al usuario conocer cuantas mutantes se han generado, cuantas no han sobrevivido y el porcentaje de cobertura que dan los casos de prueba que se han ejecutado. Mientras que en la **Figura 7** muestra los resultados más específicos, mostrando las diferentes mutaciones realizadas, marcando en rojo las líneas en las que los cambios no han sido detectados por las pruebas y en verde las que si están bien cubiertas.

2.3 Limitaciones de las herramientas analizadas y motivación

En el apartado anterior (2.2) he mostrado algunas de las herramientas existentes para realizar mutaciones automatizadas y poder analizar sus resultados con respecto a códigos Ruby. De forma general se ha mostrado como todas ellas siguen unos patrones similares con la única diferencia del número de operadores de mutaciones distintos que tienen implementados. En el caso de la herramienta vista en **2.2.1**, el único operador que estaba habilitado era el que permitía generar mutantes gracias a la eliminación de líneas de código. Por otro lado, en los casos de las herramientas vistas en **2.2.2** y **2.2.3** el número de operadores de mutación era mucho más amplio y variado, pero sin dejar de ser dos herramientas que realizan las mismas tareas y permiten prácticamente las mismas acciones por parte del usuario, exceptuando pequeñas diferencias como que en la **2.2.3** se puede indicar si una línea no se desea que sea mutada mediante un comentario.

Analizando todas estas herramientas, se pueden observar tras un pequeño tiempo de estudio, cuales son todas sus limitaciones.

Una de las limitaciones más importantes que tienen es que no permiten al usuario seleccionar que tipo de operador de mutación se desea aplicar para generar las mutaciones. De este modo las herramientas se encargan de aplicar todos los operadores posibles sin dar ningún tipo de elección. Esto limita mucho las posibilidades de estas herramientas y dan poca opción de configuración al usuario, que puede tener distintos intereses de mutación dependiendo de los proyectos o códigos que esté desarrollando.

Además, hemos visto que tanto la herramienta presentada en **2.2.2** como la presentada en **2.2.3**, que son las más completas, muestran de forma genérica los resultados obtenidos, pero no realizan una muestra detallada de esos resultados para cada mutante. Es decir, permiten al usuario conocer cuantos mutantes se han generado, cuantos han sobrevivido, cuantos no, etc. Pero no indican de forma rápida y concisa que mutantes son los que han sobrevivido, que líneas son las que se han mutado, cuales han muerto, … Para poder visualizar esto, el usuario necesita recorrer toda la salida por el terminal e ir visualizando entre los diferentes colores que mutante se ha realizado y ver si ha sobrevivido o ha muerto. Esto creo que es una forma poco usable para el usuario de conocer en detalle en que ha consistido cada mutación y su resultado.

Otro problema es la falta de una GUI que permita al usuario realizar todas sus operaciones de una forma más fácil e intuitiva, sin necesidad de tener que aprender comandos. Además, la falta de esta provoca que la visualización de los resultados sea engorrosa como ya he mencionado y que haga del análisis de los mismo, por parte del usuario, una tarea poco agradable y tediosa en las ocasiones en las que las mutaciones sean de miles de líneas.

Se ha detectado que todas estas herramientas proporcionan un conjunto cerrado de operadores de mutación y no son herramientas extensibles. De este modo el usuario debe valerse de los operadores de mutación proporcionados y no tiene la posibilidad de generar los suyos propios en caso de necesidad.

Por estos motivos se ha decidido realizar una aplicación que permita la generación automática de mutaciones en Ruby y el análisis de sus resultados. Para tomar los mejores aspectos de las herramientas ya existentes y mejorarlas. Permitiendo a los usuarios poder seleccionar los operadores de mutación que desean aplicar, con una visualización de resultados específicos más preciso que haga el estudio de estos una tarea más sencilla y agradable. Además de contar con una GUI que permita que todas las tareas sean más intuitivas e integre una mejor disposición de los resultados para ser mostrados al usuario. Además, la aplicación deberá permitir a los usuarios poder generar sus propios operadores de mutación, para cubrir aquellos casos específicos que puedan surgir en algunas ocasiones. Proporcionar al usuario un mecanismo para generar sus propios operadores de mutación puede ayudarlo a configurar la herramienta en casos particulares que de otro modo no podría probar. Por lo que permitir que la herramienta sea extensible, ayudará a los usuarios a poder realizar análisis más exhaustivos de sus casos de prueba y códigos.

2.4 Tecnologías usadas

En este apartado se proceden a presentar las tecnologías de más peso que han sido utilizadas para llevar a cabo este proyecto.

2.4.1 Ruby

Ruby [5][12] se trata de un lenguaje de programación, de propósito general, que permite desarrollar aplicaciones en ámbitos muy distinto. Es un lenguaje interpretado, por lo que no es compilado, siendo el intérprete de Ruby el encargado de analizar el código proporcionado y de transformarlo a lenguaje máquina. Es un lenguaje de programación dinámico, lo que significa que la comprobación de tipos se hace en tiempo de ejecución. Esto hace que sea un lenguaje muy flexible con soporte a muchas características avanzadas, como por ejemplo la capacidad de realizar modificaciones a un programa en tiempo de ejecución (llamada meta-programación). Además, se trata de un lenguaje open source o de código libre que puede ser interpretado en diferentes sistemas operativos y usado de forma gratuita por cualquier desarrollador.

En Ruby, existen distintas librerías para realizar pruebas unitarias. Una de ellas es test/unit. Para definir un conjunto de casos de prueba basta con generar un o varios ficheros Ruby con un requiere del fichero que contiene las funciones que se quieren probar y otro requiere de test/unit. Finalmente, se debe construir una clase que herede de Test::Unit::TestCase y en su interior contener las funciones que prueben cada funcionalidad específica del fichero requerido, realizando las comprobaciones mediante el uso de asserts. En la **Figura 8** puede verse un ejemplo.

```ruby
require_relative '../ejem1'
require 'test/unit'

class TestDeMates < Test::Unit::TestCase
  def test_sum_ok
    assert_equal  expected 2, sum( a 1, b 1)
  end
  def test_sum_bad
    assert_not_equal  expected 3, sum( a 1, b 1)
  end
  def test_return_true
    assert_boolean(sum( a -1, b -1))
  end
end
```

Figura 8 - Ejemplo fichero con pruebas usando test/unit

Su sintaxis clara y sencilla, junto con tratarse de un lenguaje de alto nivel, provoca que sea una herramienta muy útil para desarrollar códigos claros con facilidad.

2.4.2 Parser

Parser [6] se trata de una gema Ruby, programada en este mismo lenguaje, que permite parsear cualquier código escrito en Ruby para generar su AST. El AST se trata del árbol sintáctico abstracto que contiene mediante una representación de nodos y ramas, toda la información sintáctica acerca del código que representa.

El AST es una parte fundamental de las pruebas de mutación. Nos otorga el mejor método para poder obtener toda la información sintáctica de un código dado y poder trabajar sobre ella de una forma sencilla y eficaz.

Al tratarse de una pieza fundamental, su elección fue clave para poder realizar el proyecto de una forma eficiente. Parser, permite obtener el AST de una forma sencilla a partir de un código proporcionado y brinda los elementos necesarios para poder recorrerlo con facilidad y realizar modificaciones de una forma clara y rápida.

Además, cuenta con una gran variedad de tipos de nodos, que permiten cubrir todas las necesidades del lenguaje Ruby.

En la **Figura 9** podemos observar un pequeño ejemplo de obtención de un AST a partir de un código muy simple.

```
p Parser::CurrentRuby.parse("2 + 2")
# (send
#   (int 2) :+
#   (int 2))
```

Figura 9 - Obtención AST

Si observamos en detalle la **Figura 9**, mediante la función parse(), que nos ofrece la clase CurrentRuby de la gema Parser, se puede obtener a partir de una cadena que pertenezca a código Ruby, una representación de este mismo código en forma de AST. En este ejemplo el código introducido para ser transformado es "2 + 2". De este modo los nodos quedan representados como la agrupación de elementos entre paréntesis. En este caso sencillo, la transformación nos devuelve tres nodos: (int 2), (int 2) y (send (int 2) :+ (int 2)). Cada nodo contiene de forma obligatoria un tipo, en este caso los tipos que tenemos son '*send'* e '*int'* y puede contener un valor, como es el caso de (int 2), que contiene el valor '*2'*, un operador, como es el caso de (send) con '*:+'* y finalmente puede contener otros nodos hijo como es el caso, otra vez, de (send).

En la **Figura 10**, podemos observar la representación del código "2 + 2" en formato AST de una forma más visual.

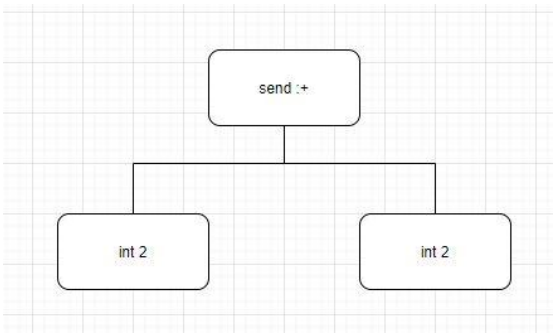


Figura 10 - AST salida grafo

Además, como ya hemos dicho, Parser ofrece al programador las herramientas necesarias para poder realizar modificaciones en los nodos y poder generar así variaciones de los códigos de entrada. Para esta tarea, nos ofrece la clase Rewriter.

El hecho de contar con esta clase hizo que se tomara a Parser como el candidato a realizar estas tareas en el proyecto, ya que con ellas los procesos de transformaciones en el código iban a quedar más legibles.

En el capítulo 3 de diseño, se abordará más en profundidad toda la funcionalidad que necesitamos para realizar las mutaciones en el proyecto y que Parser nos ofrece. Y en el capítulo 4 de desarrollo se mostrará la clase creada para realizar las modificaciones del código, que hereda de la clase Rewriter.

Finalmente cabe mencionar que se trata de una gema que desde el 30 de marzo de 2013 se encuentra en constante crecimiento y con mantenimiento continuo. Siendo la última actualización recibida dos días antes de escribir estas líneas. Esto nos garantiza que está mantenida en la actualidad, lo que nos permitirá trabajar con ella de forma segura durante todo el ciclo de vida del software que se va a desarrollar en este proyecto.

2.4.3 FxRuby

FxRuby [7] se trata de otra gema de Ruby, que contiene una librería para desarrollar interfaces gráficas de usuario para aplicaciones Ruby de una forma robusta y multiplataforma.

Se basa en el FOX Toolkit, una librería open source muy popular de C++. Esto permite realizar una programación sencilla y amable en código Ruby, pero aprovechando todas las ventajas y funcionalidades características de C++.

Además, cuenta con un sistema de instalación sencillo mediante el comando '*gem install fxruby'*. Esto es una ventaja muy grande respecto a otras librerías que existen para realizar interfaces gráficas en Ruby como Shoes o gtk3, que requieren de un proceso de instalación más complejo.

FxRuby se trata de una librería que se encuentra en mantenimiento desde su lanzamiento en 2004. Esto nos permite conocer que no estamos trabajando con una librería ya obsoleta y el hecho de que lleve en funcionamiento tantos años nos transmite seguridad.

Esta librería permite generar todo tipo de elementos gráficos como ventanas, labels, textBox, buttons, tables, checkboxs, etc. De este modo tenemos todos los elementos que necesitamos para poder montar una interfaz que permita al usuario realizar y visualizar los resultados de una forma intuitiva y sencilla.

2.4.4 PostgreSQL

PostgreSQL [15] es un sistema de gestión de bases de datos relacionales, orientada a objetos y de código abierto.
Esta herramienta nos permitirá gestionar los usuarios de la aplicación y almacenar en una base de datos todos los registros que se realicen para en un futuro poder volver a ser utilizados mediante una opción de login.

3 Diseño

En los siguientes apartados se mostrará el proceso seguido durante la realización de este proyecto para realizar el diseño de la aplicación desarrollada.

3.1 Aspectos Generales

El objetivo del proyecto, como ya se ha mencionado, es realizar una aplicación que brinde a los desarrolladores de código Ruby de una herramienta capaz de realizar pruebas de mutación de manera automatizada y que permita a los usuarios poder añadir sus propios operadores de mutación, mostrando los resultados de estas tareas de una forma sencilla y de fácil legibilidad para el usuario.

Para ello se comenzó por estudiar las aplicaciones ya existentes que se encontraban disponibles y se extrajo de ellas todos los aspectos que podían ser mejorables y que ya hemos visto en el **apartado 2.3** de este documento.

Se realizó un análisis de requisitos que se puede ver completo en el **Anexo F**, pero de forma resumida, las funcionalidades básicas con las que debe contar la aplicación son: **1)** generación de mutantes de forma automatizada, **2)** ejecución de pruebas sobre mutantes y obtención de resultados también automatizados, **3)** creación de nuevos operadores de mutación, **4)** eliminación de operadores de mutación creados anteriormente y **5)** representación de datos y funcionalidades mediante el uso de una GUI.

Así pues, el objetivo de forma general se puede dividir en dos grandes grupos. Por un lado, dotar a la aplicación de todas las funcionalidades necesarias para poder realizar las tareas que deseamos y, por otro lado, hacerla contar con un interfaz gráfico sencillo, pero que permita a los usuarios realizar sus tareas y visualizar sus resultados de una forma agradable e intuitiva.

Por este motivo la aplicación se divide en dos grandes componentes a los que llamaremos Backend y Frontend. Aunque estos términos son usados generalmente en el desarrollo de aplicaciones web, en nuestro caso se ha considerado una buena terminología, aunque nuestra aplicación no pertenezca a este grupo. Esto se debe a que nos permite encapsular de una manera sencilla en dos grandes componentes todos los elementos que necesita nuestra aplicación y ayuda a su mantenibilidad ya que consigue que los elementos que realizan tareas diferentes se mantengan desacoplados. Además, nos ofrece realizar una separación clara de los elementos en función a su tipo de lógica, lo que ayuda a mejorar la escalabilidad de nuestra aplicación y nos da opción a poder reutilizar ciertos elementos de manera sencilla.

De este modo en el Backend se encapsulará toda la funcionalidad necesaria por parte de la aplicación y en el Frontend se agruparán todos los elementos necesarios para que la aplicación cuente con una GUI.

Con esta estructura general del proyecto, las tareas a realizar se separan en dos grupos según su lógica y se facilita de esta forma su análisis y diseño.

De forma general, por tanto, el Backend proveerá al Frontend de todas las funcionalidades necesarias y se encargará de realizar todas las tareas de forma interna, mientras que el

Frontend solo deberá de encargarse de mostrar las tareas realizadas por el Backend al usuario y permitir a este acceder a las funcionalidades que el Backend le ofrece para que pueda realizar sus tareas.

A esta estructura básica hay que añadir una base de datos PostgreSQL que es el elemento clave que permitirá realizar un uso de la aplicación en el modo gestión de usuarios si quien la esté usando así lo desea. Esta base de datos únicamente se usará para almacenar usuarios creados y permitirá validar el acceso de cada usuario ya registrado mediante un nombre y una contraseña. De este modo el Backend será el encargado de realizar las distintas operaciones sobre la base de datos que hayan sido seleccionadas por el usuario en el Frontend y gestionar los datos devueltos cuando se realice una query sobre la misma.

Con la inclusión de la base de datos, en la **Figura 11** se puede observar el diagrama con los aspectos generales y la interacción entre los dos grandes grupos, el usuario y la base de datos.

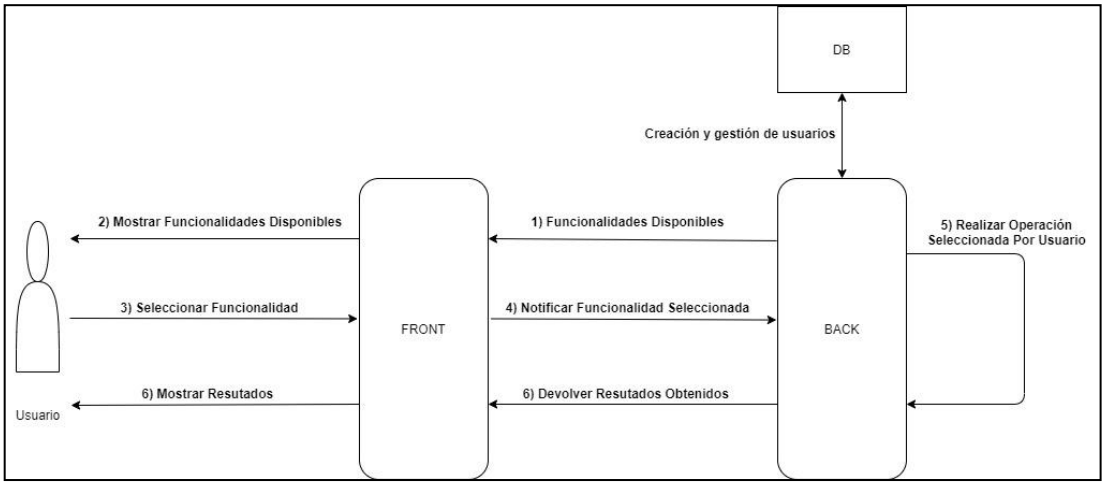


Figura 11 - Diagrama Estructura General Con DB

Finalmente hay que mencionar que, si la base de datos está configurada, los usuarios podrán acceder tanto como usuarios registrados o como con un modo de usuario sin registrar, mientras que, si no está configurada, solo se podrá acceder en el modo de usuario sin registrar. El modo en el que se acceda a la aplicación generará unas características especiales a la hora de visualizar, crear y eliminar operadores de mutación que veremos más adelante.

3.2 Backend

Una vez estudiada la vista general del proyecto y su separación en los dos grandes grupos, junto con los requisitos de este, se comenzó a realizar un análisis más detallado de cada uno de ellos por separado.

3.2.1 Carga De Operadores

Los operadores son una de las piezas fundamentales de la aplicación. Estos permiten a los usuarios conocer que operaciones de mutación pueden aplicar sobre sus ficheros y permiten al sistema conocer la mutación correcta que debe realizar.

El sistema debe contar de inicio con un conjunto de operadores que sean de carácter general para Ruby. Esto quiere decir, que se trata de operadores que se consideran importantes para el lenguaje y que su necesidad de uso suele ser algo común.

Por este motivo los operadores con los que contará por defecto el sistema son los que se pueden visualizar, junto con una descripción, en la tabla del **Anexo B** de este documento. Se han implementado operadores básicos para Java que también son aplicables a Ruby extraídos del documento [16] y operadores de mutación específicos para Ruby extraídos del documento [17].

Se decidió que los operadores que trae por defecto la aplicación y los nuevos operadores que se crearan debían ser almacenados en un fichero con extensión .csv. Esto nos permite desacoplar por completo el uso de la aplicación de la necesidad de una base de datos, que de otro modo, el usuario se vería obligado a configurar antes de realizar la primera ejecución al tratarse de una aplicación local y en la que cada instalación requeriría de su propia base de datos.

Así pues, esta decisión permite a la aplicación ser capaz de autogestionar la carga de los operadores del fichero .csv a la aplicación de forma automática cada vez que se arranque la aplicación o este fichero sea actualizado, y hace que el usuario pueda desentenderse de este asunto y en caso de no requerir el modo de gestión de usuarios, poder usar la aplicación únicamente instalando una gema y sin necesidad de configurar nada.

El fichero csv, debe contar con la siguiente cabecera:

```
name,operator,new_operator,operation_type,description,user
```

Esta es necesaria para poder tener toda la información requerida por parte de la aplicación y poder gestionar correctamente los operadores. De este modo, el campo *name* indica el nombre del operador, *operator* indica el operador actual que debe de encontrarse en el sistema, *new_operator* indica el nuevo operador que debe ser incluido en la mutación en lugar del operador indicado en el campo *operator*, el campo *operation_type* permite conocer el tipo de operación y este puede contener los valores 'R', 'A', 'D', 'R_U'. El tipo 'R' especifica que se trata de una operación de reemplazo, el tipo 'A' especifica que se trata de una operación de agregación. En este caso el campo *operator* tendrá valor vacío (''). El tipo 'D' especifica que se trata de una operación de eliminación. En este caso el campo *new_operator* tendrá valor vacío (''). Y finalmente el tipo 'R_U' especificara que se trata de una operación de reemplazo creada por un usuario.

Continuando con los campos, el campo *description*, contiene la descripción del tipo de mutación que genera ese operador y finalmente el campo *user*, contiene la información de que usuario ha creado dicho operador. Si el operador ya venía como base del sistema, este campo valdrá 'default', si lo ha creado un usuario registrado entonces este campo tiene el valor del nombre del usuario que lo creó y si ha sido creado por un usuario usando el modo sin registrar, entonces contendrá como valor 'default_name'.

En la **Figura 12** se puede ver un ejemplo del contenido de este fichero .csv.

```
1  name,operator,new_operator,operation_type,description,user
2  "sum","+","-","R","Change + with -","default"
3  "irange_to_erange","..","...","R","Replace range with .. to ... .Ex a[1..5] -> a[1...5]","default"
4  "delete_!","!","","D","Delete the negation of the code.","default"
5  "add_!","","!","A","Add the ! to all elements they may contain.","default"
6  "erange_to_irange","...","..","R_U","Replace range with ... to .. .Ex a[1...5] -> a[1..5]","default_name"
7  "res","-","+","R_U","Change - with +","pepe"
```

Figura 12 - Fichero csv con la definición de operadores de mutación

La aplicación por tanto se encargará de leer este fichero y cargar los operadores en el sistema solo al arrancar o cuando sufra algún cambio. Esto permitirá optimizar los tiempos de ejecución ya que solo habrá que leer de disco en esos dos casos, mientras que el resto del tiempo los datos se encontrarán cargados en memoria RAM en la cual los tiempos de escritura y lectura son mucho más rápidos.

Además, la ampliación de operadores por defecto del sistema será muy sencilla de realizar y bastará con añadir tantas nuevas líneas a este fichero como operadores de mutación nuevos se quieran agregar.

3.2.2 Usar, crear y eliminar operadores del sistema

Una vez que los operadores estén cargados en el sistema, el Backend debe de ser capaz de indicar al Frontend que operadores están disponibles para ser usados y gestionar la creación y eliminación de estos.

Los operadores en el sistema pueden clasificarse en tres grupos en función de por quien fueron creados. A continuación, se muestra una enumeración de los grupos:

1) *default*: Son aquellos que vienen por defecto en el sistema.
2) *usuario_registrado*: son aquellos creados por un usuario que ha entrado haciendo login con su cuenta.
3) *usuario sin registo*: aquellos creados por usuarios que han accedido con el modo sin usuario registrado.

A la hora de mostrar los operadores de mutación disponibles para ser usados, el sistema mostrará siempre a todos los usuarios aquellos que pertenecen al grupo 1) y además si se ha accedido mediante un usuario registrado, se mostrarán los pertenecientes al grupo 2) que contengan el mismo nombre que el usuario actualmente conectado en el sistema, y si ha accedido sin usuario registrado se mostrarán los pertenecientes al grupo 3).

En la funcionalidad de borrado, solo podrán ser borrados los pertenecientes al grupo 2) si el usuario actual que ha accedido con un usuario registrado es el creador de alguno de ellos. Mientras que los del grupo 3) podrán ser eliminados por cualquier usuario que acceda en el modo sin registro. Finalmente, en ningún caso se permitirá a un usuario, independientemente del modo en el que haya accedido, borrar un operador perteneciente al grupo 1).

Finamente los usuarios podrán crear operadores de mutación, del tipo reemplazo R_U, indicando un nombre de operador que no debe existir en el sistema con anterioridad, ni en el conjunto 1), ni en el conjunto 2) si el usuario ha accedido con un usuario registrado o ni en el conjunto 3) si ha accedido sin registro. Además del nombre deben indicar el operador actual para ser reemplazado, el operador por el que se va a reemplazar y la descripción de este.

De este modo se podrán generar operadores para reemplazar nuevos operadores de tipo aritmético, lógico, relacional, condicional o de asignación que puedan surgir en un futuro, nombre de variables, nombre de funciones en la llamada a las mismas (no es su definición) y finalmente cadenas de caracteres tanto que se encuentren entre comillas simples como dobles.

3.2.3 Generación de mutantes y ejecución de pruebas.

Una vez que el usuario ha especificado el fichero que desea mutar, el path con las pruebas unitarias y los operadores que desea aplicar, entonces hay que comenzar con el proceso de generación de mutaciones y ejecución de pruebas.

El orden para realizar estas tareas será el siguiente. En primer lugar, se extraerá el AST del código seleccionado, en segundo lugar, se cogerá el primer operador seleccionado por el usuario y se recorrerá el AST buscando los nodos o conjuntos de nodos en los que se deba realizar la mutación. En cada nodo o grupo de estos en los que haya que realizar la mutación que indique el operador, se realizará la misma y se guardará el resultado en un fichero. Cada mutación realizada será independiente de las anteriores que se hayan podido realizar y cada mutante generado se guardará en ficheros distintos. Una vez generados todos los mutantes para un operador de mutación, se ejecutarán las pruebas unitarias sobre el fichero original, si este las pasa todas, entonces se ejecutarán las pruebas sobre cada mutante y se guardarán los resultados obtenidos junto con la mutación realizada, para posteriormente ser analizados, estructurados y mostrados al usuario.

En la **Figura 13** se puede observar un diagrama de flujo de este proceso que ayudará a comprender mejor el mismo.

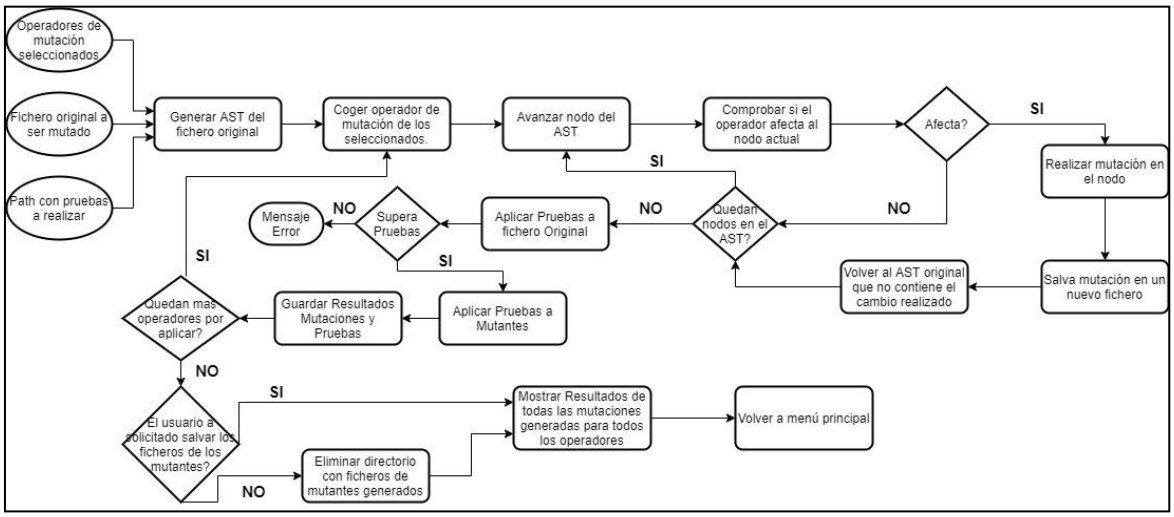


Figura 13 - Diagrama Flujo generación AST, mutantes y pruebas

Me gustaría explicar un poco más en profundidad como se realizan las mutaciones concretamente y como se generan y salvan. Para ello me voy a ayudar de la **Figura 14** que se puede observar a continuación.

Figura 14 - Generación Mutantes

En la **Figura 14** se puede observar un fichero original que es la raíz del árbol y los tres mutantes que se pueden generar a partir de él aplicando el operador de mutación 'sum' que reemplaza el operador aritmético + con el – y el operador de mutación 'mul' que reemplaza * con /. Así pues, la aplicación tomará el fichero original, como ya hemos visto, y lo transformará a AST, entonces seleccionará el primer operador de mutación seleccionado, en este caso 'sum', y comenzará a recorrer el AST. Cuando llegue al nodo (send (int 1) :+ (int 2)) comprobará que puede realizar una mutación, así que llamará a la clase encargada de realizar dicha tarea, que la veremos en el apartado de desarrollo, y generará el mutante que podemos observar como la primera hoja del árbol, empezando por la izquierda, de la **Figura 14**, guardando este resultado en un primer fichero mutacion0_0.rb donde el primer cero indica que se trata del primer operador de mutación seleccionado y el segundo que se trata de la primera mutación para ese operador.

Una vez salvado el fichero, restaurará el AST original y continuará avanzando por los nodos desde el nodo en el que se encontraba antes de realizar el cambio en el código hasta que llegue al nodo (send (int 2) :+ (int 3)), en este caso realizará los mismos pasos que antes dando como resultado la mutación observada en la hoja central de la **Figura 14**. Guardará este mutante en un nuevo fichero mutacion0_1.rb ya que sigue trabajando con el operador de mutación inicial, pero es el segundo mutante que genera a partir de él. Una vez salvada la mutación, restaurará el AST al original y lo continuará recorriendo hasta llegar al último nodo y comprobar que no puede realizar ninguna mutación más con el operador actual. En ese momento realizará las pruebas sobre el fichero origina y si las pasa, entonces ejecutará las pruebas sobre cada mutante y guardará los resultados. Tras esto pasará a seleccionar el segundo operador que le hemos facilitado, 'mul', y comenzará nuevamente a recorrer desde el inicio todos los nodos del AST procedente del fichero original sin mutaciones. Tras pasar por los nodos (send (int 1) :+ (int 2)) y (send (int 2) :+ (int 3)) y comprobar que no puede realizar ninguna operación de mutación sobre ellos con el operador 'mul', llegará al nodo (send (int 3) :* (int 4)) y comprobará que puede realizar una mutación, así que procederá a realizarla, dando como resultado la hoja más a la derecha de la **Figura 14**. Este mutante será salvado en el fichero mutacion1_0.rb ya que se trata del segundo operador de mutación que estamos usando y la primera mutación que genera con él.

Tras salvar el fichero continuará recorriendo el AST hasta terminar de pasar por todos los nodos del AST. En ese momento buscará si quedan más operadores de mutación que se hayan seleccionado por pasar, comprobará que no y por tanto procederá a ejecutar los tests sobre estas nuevas mutaciones directamente, ya no lo hará sobre el fichero original ya que se comprobó que los superaba en la primera vez que se llegó a este punto, guardará los resultados obtenidos y al no tener más mutaciones que generar procederá a mostrar los resultados obtenidos para todas las mutaciones generadas a partir de todos los operadores seleccionados.

3.2.4 Recopilación de resultados y presentación

Una vez realizadas las mutaciones y ejecutadas las pruebas sobre las mismas, se procederá a la recopilación de resultados y su organización para ser presentados.

El objetivo es lograr dar la mayor cantidad de información posible de una forma ordenada. Por este motivo se ha decidido agrupar los resultados en dos grupos en función de a quien pertenecen. Por un lado, se recopilarán resultados genéricos en función a cada operador de

mutación utilizado y por otro, se tomarán y mostrarán resultados específicos para cada mutante.

Cabe destacar que en el **Anexo D** de este documento se podrá encontrar un ejemplo con Figuras de la aplicación desarrollada de los resultados mostrados de forma genérica y los mostrados de forma específica.

3.2.4.1 Resultados Genéricos

Como ya se comentó anteriormente en esta memoria, las pruebas de mutación ayudan a detectar la calidad de las pruebas generadas. Así que el objetivo de mostrar datos genéricos para cada operador de mutación aplicado es este. Con este fin se recopilarán cuantos mutantes se han generado para cada operador de mutación y se mostrará también cuantos de ellos han sobrevivido (han logrado pasar las pruebas) y cuantos han muerto (no han logrado pasar las pruebas). También se mostrará el porcentaje de mutación que se calcula como:

$$Porcentaje\ de\ mutación = \frac{nº\ mutantes\ muertos}{nº\ total\ de\ mutantes}$$

Los datos comentados permitirán al usuario conocer el alcance de la efectividad de sus pruebas y ayudará a realizar una aproximación de la cobertura de estas.

Si el número de mutantes que han sobrevivido es 0 para un operador de mutación, implicará que el porcentaje de mutación es del 100% y por tanto el usuario podrá saber que cobertura de sus pruebas es máximo para las líneas que ha modificado ese operador y en consecuencia que ese caso está bien probado. En caso de sobrevivir algún mutante, entonces el usuario sabrá que cobertura de sus pruebas ya no es del 100% y que deberá mejorar las pruebas existentes o generar otras nuevas para cubrir esos casos que las pruebas que actualmente tiene no son capaces de detectar. Igualmente, el porcentaje de mutación permite al usuario hacerse una idea más específica de que porcentaje de mutantes han muerto al pasar sus pruebas para un operador de mutación dado. De este modo un porcentaje de mutación alto implica una mejor calidad y cobertura de los casos de prueba desarrollados, mientras que uno bajo implica una baja calidad o la falta de casos de prueba por implementar.

3.2.4.2 Resutados Específicos

Por otro lado, se mostrarán los resultados específicos para cada mutante, estos permitirán al usuario tener un conocimiento más detallado de donde provienen los datos vistos en los resultados genéricos. Por cada mutante generado se mostrará, que operador de mutación lo ha generado, el estado del mutante (si ha sobrevivido o no), la línea del fichero original que ha sido mutada, el contenido de la línea original, el contenido de la línea tras la mutación y cuantas pruebas se han ejecutado sobre este mutante, cuantas han pasado, cuantas han dado fallo y cuantas han dado error. Un error se produce cuando algo se rompe y ocurre una excepción, como una referencia de objeto nulo, mientras que el fallo se produce cuando no se cumplen los criterios de prueba. es decir, cuando assert() falla.

Estos datos permitirán al usuario conocer que mutantes específicos han sobrevivido y cuales han muerto, conociendo con exactitud que cambios han sido detectados correctamente y cuales no respectivamente.

De este modo el usuario podrá saber más fácilmente el cambio concreto que ha provocado que esa línea no fuera detectada y podrá realizar nuevas pruebas más específicas para cada caso concreto, ayudando a mejorar la cobertura y eficacia de sus pruebas.

3.2.5 Base de datos y gestión de contraseñas.

En este proyecto la base de datos tiene un objetivo muy concreto y es permitir que la aplicación soporte un modo de ejecución con gestión de usuarios en aquellos casos en los que un grupo de desarrolladores de dos o más personas utilicen un mismo entorno de desarrollo. En caso de que lo consideren oportuno, configurando la base de datos durante la instalación como se puede observar en el **Anexo A** de este documento, podrán disponer de distintas cuentas para hacer uso de la aplicación.

El hecho de tener configurada la base de datos permite a cada usuario registrado tener acceso únicamente a los operadores por defecto de la aplicación y a los creados por sí mismo, evitando que pueda eliminar operadores creados por otros usuarios de forma no deseada y reduciendo el tiempo de búsqueda de operadores a usar.

Por tanto, la base de datos, que se desarrollará en PostgreSQL, contará únicamente con una sola tabla, llamada users, que tendrá el objetivo de almacenar los usuarios registrados en la aplicación. Con este fin, contará con tres columnas: la PK que será la columna name, que almacenará el nombre del usuario y por otro lado las columnas password y salt. Estas columnas nos permitirán almacenar la contraseña del usuario de forma segura evitando guardar la contraseña en texto plano y guardándola mediante el hash de la concatenación de la contraseña y el salt aleatorio. Esto protegerá la base de datos ante el robo de esta por parte de un tercero que busque realizar ataques por tablas pre-computadas o tablas arcoíris.

En el **Anexo E** de este documento se puede encontrar una explicación más detallada del mecanismo de defensa utilizado para proteger las credenciales de los usuarios que utilicen la aplicación y que función hash y de generación de salt se ha utilizado para realizar esta tarea.

	name [PK] character varying (20)	password character varying (60)	salt character varying (30)
1	pepe	$2a$12$r0xj0qdRdMK2ZYlq5Ec4..jlRCrt5xlHksXWhGtjj.eA4nL2C39zC	$2a$12$r0xj0qdRdMK2ZYlq5Ec4..
2	maria	$2a$12$mxRQ08U66Cz1UzvnpmIKDelNzwj5cRNuB3J7G6s0O6/x0aqyf6Mee	$2a$12$mxRQ08U66Cz1UzvnpmIKDe

Figura 15 - Contenido tabla users

Finalmente, en la **Figura 15** se puede observar un ejemplo de contenido de la tabla users de la aplicación con dos registros insertados.

3.3 Frontend

Una vez analizada toda la funcionalidad interna de la aplicación y haber realizado su desarrollo se procedió realizar un análisis de los requisitos del Frontend que permitieran presentar toda la funcionalidad del Backend al usuario por medio de una GUI fácil e intuitiva de usar.

Se estructuró la GUI en cinco partes básicas. Una vista para mostrar el registro y acceso de usuarios a la aplicación, otra vista para poder crear operadores de mutación, una para poder eliminarlos, otra cuarta que permitiera seleccionar que operadores se querían utilizar para generar los mutantes de un fichero y unos casos de prueba que se pudieran seleccionar

también en esta misma vista y finalmente una última vista para poder mostrar los resultados obtenidos y el código de los mutantes generados en caso de que estos hubieran sido guardados a petición del usuario.

Tras comenzar con el desarrollo, se fue consciente también de que era necesario realizar una vista a modo de menú principal que permitiera navegar entre las cuatro primeras vistas mencionadas anteriormente.

En la **Figura 16**, se puede observar el diagrama de navegación entre las diferentes vistas que debían desarrollarse para la construcción del GUI.

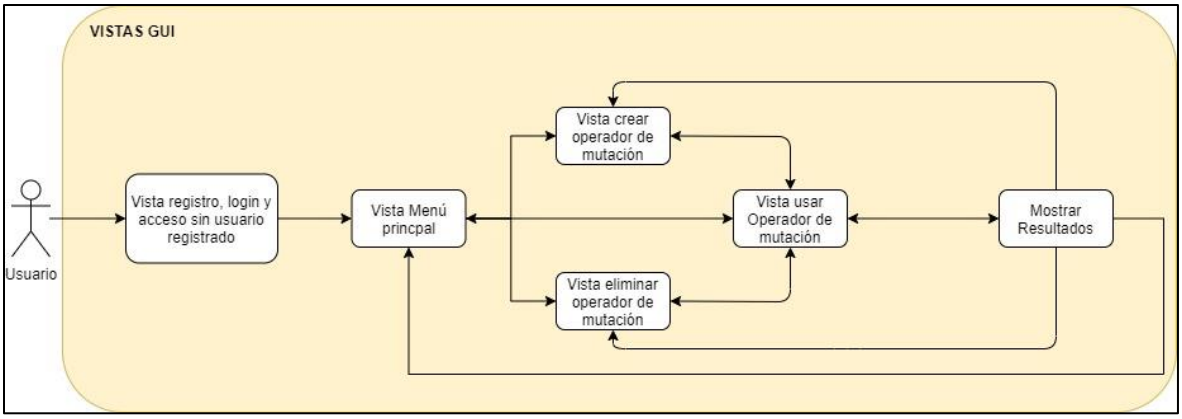


Figura 16 – Diagrama de navegación entre vistas.

Además, se decidió realizar cuatro tipos de ventanas distintas que contuvieran a estas vistas. La primera una pequeña para mostrar el progreso del arranque de la aplicación, que, aunque en la actualidad es inmediato, en un futuro si se hace extensible a otros lenguajes, puede tener una carga importante de datos y su tiempo de carga puede ser notorio, de este modo ya queda implementada. Por otro lado, una ventana un poco más grande para mostrar la vista de registro y acceso de usuarios. En tercer lugar, una pequeña ventana que debe mostrarse únicamente cuando el programa está realizando los mutantes y ejecutando las pruebas para que el usuario sepa que la aplicación está trabajando y no se ha quedado colgada y por último la ventana principal más grande de todas que mostrará las vistas de selección, creación y eliminación de operadores junto con la vista de mostrado de resultados.

Aprovechando la case FXMessageBox que nos brinda Fxruby, se programarán pequeñas ventanas secundarias de información o aviso.

En las **Figuras 17 y 18** se pueden observar algunas maquetas realizadas.

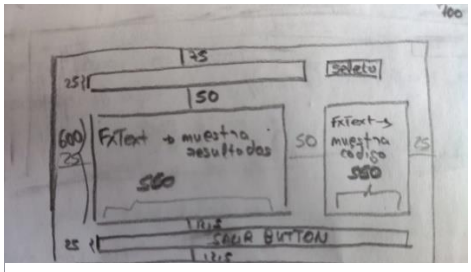


Figura 17 – Maqueta diseño de vista para mostrar resultados

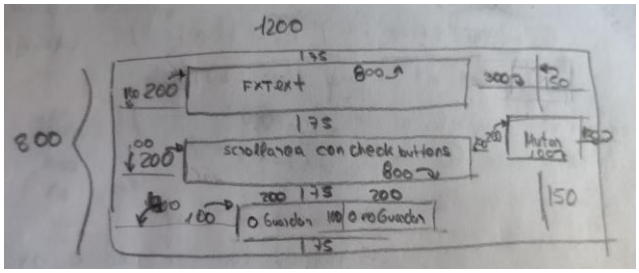


Figura 18 – Maqueta para diseño de vista de selección de operadores de mutación, fichero a mutar y pruebas

4 Desarrollo

En esta sección del documento se proceden a mostrar los aspectos más relevantes del desarrollo de la aplicación tanto para la parte de funcionalidad interna como para la GUI.

En el enlace https://gitlab.com/ivan_diaz_moreno/tfg_en_ruby-pruebas_mutacion_ruby/-/tree/feature/app_gui_users se puede encontrar el repositorio con todo el código de la aplicación desarrollada. La rama *feature/app_gui_users* es la que contiene la última versión.

En las **Figuras 19 y 20** se muestra la estructura interna del proyecto de forma general y la estructura de Backend.

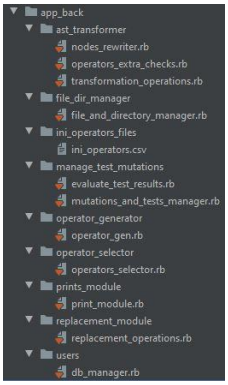


Figura 19 - Estructura Backend

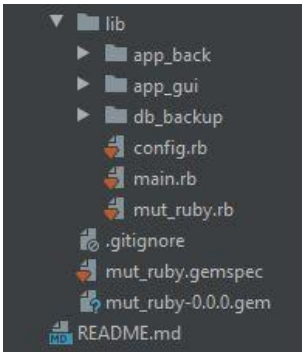


Figura 20 - Estructura General

4.1 Backend

En esta sección se proceden a mostrar las partes con mayor importancia del desarrollo de la funcionalidad interna de la aplicación.

Los ficheros main.rb y mut_ruby.rb que se pueden observar en la **Figura 20** son los ficheros encargados de poner en marcha la aplicación. mut_ruby.rb tiene el nombre de la gema (paquete de librerías que se instalan en el Sistema para poder ser usadas) para que sea el archivo de inicio cuando esta sea llamada. Este se encarga de ejecutar el fichero main.rb, mediante un comando en la Shell, que arranca la aplicación. El fichero config.rb contiene la extracción de los paths necesarios durante el uso de la aplicación, para que de esta manera la aplicación pueda correr en distintos entornos locales sin la dependencia de paths estáticos, además de la configuración del usuario y contraseña con el que conectarse a la base de datos que debe haber sido configurada, como se indica en el **Anexo A**, si desea utilizarse.

El fichero file_and_directory_manager.rb contiene la funcionalidad necesaria para trabajar con paths y ficheros, por lo que contiene funciones para evaluar la extensión de un fichero, la existencia del mismo o de un path, creación de carpetas, eliminación de las mismas, obtención de ficheros de un directorio, obtención del número de líneas de un fichero y un pequeño manejador de contexto creado personalmente para que se encargue de abrir ficheros, operar con ellos y cerrarlos automáticamente cuando termine con ellos para evitar olvidar cerrarlo, ya que en esta aplicación la apertura y cierre de ficheros a la hora de generar mutantes es bastante alta. Este manejador de contexto realizaría una función similar al manejador de contexto ‘with open()’ de Python.

4.1.1 Carga, generación y eliminación de operadores

El fichero operator_gen.rb contiene una clase que es la encargada de gestionar todas las operaciones que interactúan con el fichero .csv que contiene los operadores de mutación de la aplicación. De este modo se encarga de leer el contenido del fichero .csv y cargar los operadores en la aplicación, escribir en este archivo para almacenar los nuevos operadores que se crean y borrar de él aquellos operadores que son eliminados por parte de los usuarios.

Cada vez que se produce una operación de inserción o de borrado, siempre posteriormente, se llama a la función *load_default_operators()* de esta clase para que cargue los nuevos cambios producidos en el fichero .csv a la aplicación.

4.1.2 Selección de operadores

La clase Selector, que se encuentra en el fichero opreators_selector.rb del directorio /operator_selector, me permitió poder realizar todas las selecciones relacionadas con la generación de mutantes y selecciones en la interfaz de una forma no gráfica. De este modo pude: seleccionar que operadores de mutación quería utilizar, seleccionar el path que contiene las pruebas que desea aplicar y seleccionar el fichero que se desea mutar.

Se ha debido gestionar el control de la existencia de rutas y ficheros a la hora de seleccionar estos mismos en esta clase para evitar la introducción de datos erróneos.

Con la posterior construcción de la GUI estas funciones dejaron de usarse y únicamente se utilizan las funciones que permiten conocer que operadores de mutación se encuentran cargados en el Selector y la función *get_operator_and_new_operator()*, que dada una lista de nombres de operadores de mutación, guarda en una lista el elemento actual que hay que buscar en el código y el nuevo por el que hay que reemplazarlo asociadas a ese nombre de operador.

Aunque como hemos dicho este modo de ejecución no gráfico ya no se usa, todas estas funciones y el main que lo hacían funcionar continúan en el código por si en un futuro se desea volver a usar este modo durante nuevas fases del desarrollo.

4.1.3 Generación de AST

Una vez que el usuario ha seleccionado los operadores que desea aplicar, el fichero que desea mutar y el conjunto de casos que desea validar, se procede a realizar la obtención del AST del código de entrada.

La funcionalidad necesaria para poder realizar esta tarea la encontramos en el fichero transformation_operations.rb que se encuentra en el directorio /ast_transformer. Esta funcionalidad comprende leer el contenido del fichero de entrada y generar el AST a partir de él.
En la **Figura 21** se puede observar el código desarrollado que permite generar el AST a partir del contenido leído del fichero seleccionado por el usuario.

```ruby
def generate_ast_from_content_file
  # Given the content of a read file saved in @content transform it into ast
  # :return: ast of the code and the buffer with the source code information,
  # the error if the content has errors or nil if there are not content
  if @content != -1
    begin
      @ast = Parser::CurrentRuby.parse(@content)
    rescue Exception => error
      print "ERROR -> no se puede generar mutante ya que el codigo proporcionado contiene errores: ", error, "\n"
      return error
    else
      # A buffer with source code. Buffer contains the source code itself,
      # associated location information (name and first line), and takes care of encoding.
      # A source buffer is immutable once populated.
      code_source_buffer = Parser::Source::Buffer.new( name 'buffer')
      code_source_buffer.source = @content
      return @ast, code_source_buffer
    end
  else
    puts 'No se ha realizado transformacion ya que no hay contenido leido'
    return nil
  end
end
```

Figura 21 - Código obtención AST

Este es el paso previo a la generación de mutantes y por tanto a la modificación del contenido del AST que estamos generado. La gema Parser que se ha utilizado para realizar esta tarea como ya se ha comentado en secciones anteriores de este documento, permite generar el AST de un código dado con formato string mediante el uso de su función *parse()*, la cual devuelve un objeto del tipo Parser::AST::Node que contiene el nodo raíz del AST generado. En la **Figura 22** se pude ver un ejemplo de este proceso de transformación, en el que dado un fichero de entrada se muestra el objeto Parser::AST::Node devuelto que contiene toda la información del árbol generado para ese código.

Figura 22 - Generación de AST a partir de código de entrada

Como podemos observar en la **Figura 22**, parser nos permite tener toda la información del código en formato AST. Nos genera todo el árbol y nos devuelve el nodo raíz de este. De este modo nos encontramos en disposición de comenzar a recorrerlo para poder explorar sus hijos que se encuentran en el atributo children. Estos hijos pueden ser terminales como es el caso del nombre de la función :sum, o pueden ser nuevos nodos del tipo Parser::AST::Node que contenga a su vez nueva información en su interior con nuevos hijos para explorar. Además, podemos conocer el tipo del nodo actual en el que nos encontramos gracias a su atributo 'type'.

Además la función *parse()* es capaz de detectar errores sintácticos en el código de entrada, por lo que es una herramienta muy completa que nos permite avisar al usuario en caso de que el código que nos esté proporcionando contenga errores de este tipo y así parar el proceso de mutación antes de que pueda causar otros problemas.

Además del AST este proceso también genera un buffer del tipo Parser::Source::Buffer, que almacenará el contenido del fichero a partir del cual se ha generado el AST ya que posteriormente nos será necesario para poder realizar las modificaciones en los nodos del AST y recuperar el estado original del árbol tras cada modificación.

4.1.4 Proceso de modificación del AST y generación de mutantes

Una vez que tenemos el AST del código proporcionado por el usuario, es el momento de comenzar a recorrerlo y generar los mutantes. El proceso que se quiere desarrollar es el explicado en la fase de diseño. Coger el primer operador de mutación que haya seleccionado el usuario, recorrer el AST buscando todos los nodos en los que se puede ser aplicado y realizar el cambio en cada uno de ellos, salvando el resultado en un fichero independiente. Una vez que se ha recorrido el árbol completo, se ejecuta el conjunto de casos de prueba sobre el fichero original y si no salta ningún error entonces se ejecuta sobre los mutantes generados y se guardan los resultados. Después se procede a coger el siguiente operador de mutación seleccionado y se repite el proceso solo que en las sucesivas ocasiones ya no se ejecutan las pruebas sobre el fichero original. En caso de que los casos de prueba detecten errores en el fichero original se cortará el proceso y se notificará al usuario.

Los ficheros que contienen toda la funcionalidad necesaria para realizar esta tarea son: replacemente_operations.rb, nodes_rewriter.rb y operators_extra_checks.rb.

El fichero replacemente_operations.rb, es el encargado de recorrer el árbol, guardar los mutantes generados en ficheros y llamar a las pruebas para que se ejecuten cuando corresponde almacenando los resultados obtenidos de estas. El árbol se recorre en profundidad post-orden, es decir se recorre primero el hijo izquierdo y todos sus subhijos, luego el hijo derecho y todos sus subhijos y finalmente el nodo padre. Para ello, se hace uso del atributo children de cada nodo, que contiene una referencia a todos sus hijos.
Se decidió realizar este modo de recorrer el árbol para generar primero los mutantes más específicos del código y después los más genéricos que involucraran más nodos. En el **Anexo H** de este documento se puede encontrar una descripción del procedimiento de recorrido de un árbol en modo postorden. Mientras que los ficheros nodes_rewriter.rb y operators_extra_checks.rb son los ficheros encargados comprobar si el operador de mutación actual es aplicable a un nodo y en caso de serlo realizar la modificación del AST.

Para poder realizar las modificaciones en el fichero AST comprobé que la gema parser que estaba utilizando para generar el AST incluía una clase que permitía modificarlos. Esta clase se llama TreeRewriter y contiene un método llamado *rewrite()* que llama al método especifico de reescritura de cada nodo en función a su tipo y es capaz de generar como salida el string con los cambios realizados en el AST, si los hay, y mantener el AST sin alteraciones para poder seguir siendo recorrido sin que los cambios realizados aparezcan en cambios futuros. Estos métodos específicos de reescritura deben ser definidos con la sintaxis *on_tipoDeNodo()* en la clase que herede de TreeRewriter. De este modo la función *rewrite()* sabe a que función debe pasar el nodo actual en función al tipo que contenga. Por ejemplo, para un nodo tipo (arg :a), al realizar la llamada *rewrite(**(arg :a)**)*, esta llamará a la función *on_arg(node)*, la cual está especializada para poder evaluar de forma específica los nodos de tipo arg y si el operador de mutación actual es aplicable al nodo. En caso de serlo, realizará la operación de reemplazo mediante las funciones *replace(range, content)*, que permite cambiar un elemento por otro nuevo conociendo el Parser::Source::Range del primero, que son las coordenadas internas que utiliza la gema parser para ubicar cada elemento en el árbol, y *wrap(range,before,after)*, que permite agregar elementos antes y después de un elemento dado del cual se conoce su Parser::Source::Range.

Por tanto todas las funciones específicas del tipo on_nodeType() que se necesitan para tratar los distintos tipos de nodos, se han definido en el fichero nodes_rewriter.rb que contiene la clase RewriteTree que hereda de la clase TreeRewriter de la gema parser. En ella, se definen

todas las funciones para cada tipo de nodo y se realizan las comprobaciones necesarias para ver si un operador de mutación es aplicable y en caso de serlo realizar los cambios oportunos.

Esta misma clase contiene un método *replace_current_operators(current_operators)* que permite especificar a la misma cual es el operador de mutación actual que se quiere aplicar.

Algunas mutaciones requieren de comprobaciones demasiado largas como para ser incluidas directamente en la clase RewriteTree que hemos mencionado, como es por ejemplo el caso de agregar el operador de negación lógico (‘!’) a nuestro código o el operador (‘~’) entre otros. Por este motivo se creó el módulo ExtraChecks que se encuentra en el fichero operators_extra_checks.rb y que contiene la funcionalidad necesaria para poder realizar todas las comprobaciones necesarias que este tipo de operaciones de mutación requieren.

A continuación, vamos a evaluar dos ejemplos de mutación, uno que no requiere de funcionalidad del módulo ExtraChecks y otro que sí, aunque en realidad hay tantos casos distintos para cada operador por defecto implementado y tipo de nodo, que en el código desarrollado se pueden encontrar, pero que comentarlos uno a uno en esta memoria es imposible debido a la limitación de espacio.

En la **Ilustación 26** podemos observar la función encargada de tratar los nodos de tipo irange, que son aquellos que contienen la información de acceso a un conjunto de un iterable. Un ejemplo sería iter[1..5] que en AST estaría codificado como ‘(irange (int 1) (int 5))’.

```
def on_irange(node)
  if @@operators_to_apply['type'] == "R"
    # Check if is a replace of .. with ...
    if @@operators_to_apply['operator_to_replace'] == ".." && @@operators_to_apply['new_operator'] == '...'
      replace(node.loc.operator, @@operators_to_apply['new_operator'])
    end
  end
end
```

Figura 23 - Función on_irange

Como podemos observar en la **Figura 23**, se comprueba el tipo de la operación de mutación actual y en caso de ser de reemplazo y que busque reemplazar el elemento ‘..’ por el elemento ‘…’, obtiene el Range de ‘..’ que en este tipo de nodo se encuentra almacenado en el campo location.operator (loc.operator en el código) y realiza un replace del contenido de esa posición en el nodo con el nuevo contenido ‘…’ devolviendo un string con los cambios realizados y restableciendo el contenido del AST al original una vez que el cambio ha sido devuelto como string.

Por ejemplo, para el AST:

(def :fun_a (argr (arg :a) (arg :b)) (begin (lvasgn :a (send (lvar :b) :[]
(irang (int 2)(int 5))))(return (lvar :a))))

Esta función, cuando sea llamada al llegar al nodo *(irang (int 2)(int 5)),* devolvería el string “def fun_a(a,b) a = b[2…5] return a end” con el cambio de ‘.. ‘por ‘…’ pero el AST se habría restablecido al original tras realizar el cambio gracias al Buffer que generamos en la creación del AST que contiene el valor original del código y que internamente utiliza el método rewrite*()* para reestablecer el AST tras realizar el cambio.

Veamos ahora el segundo ejemplo. En la **Figura 24** se puede observar el método on_send() que es el tipo de nodo que contiene o debe contener los operadores de negación y que por tanto es el encargado de evaluar si se pude agregar una negación en el nodo actual o no. Aunque también es el encargado de evaluar si se puede reemplazar un operador aritmético, si hay un comprobador de instancia a reemplazar como *is_a?* con *instance_of?*, si existe el método freeze para poder ser eliminado y si se trata de un acceso a un iterable mediante índices y se quiere realizar la operación del cambio de índice de positivo a negativo o al contrario.

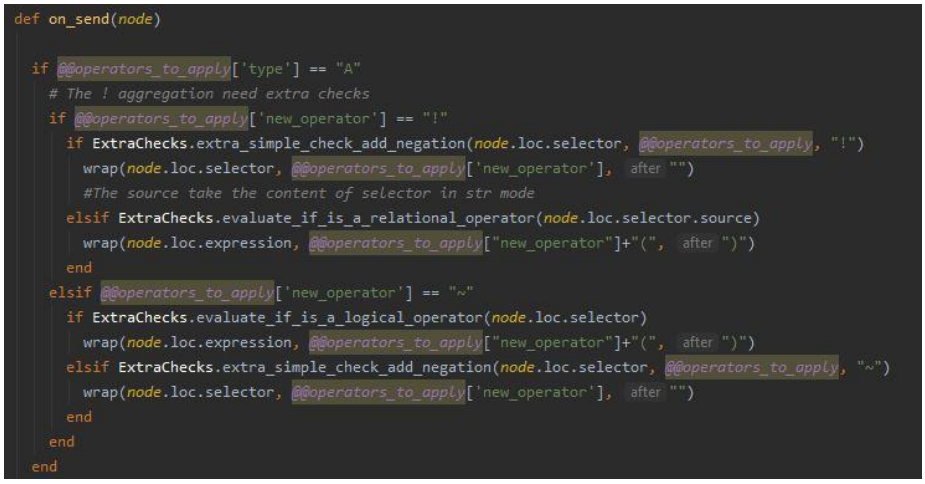


Figura 24 - Parte del método on_send que muestra el proceso de comprobación de si se puede agregar una negación y realizar la misma en caso de que se pueda.

Este método que se observa en la **Figura 24** es mucho más largo, pero se ha recortado en la imagen para mostrar solo la parte encargada de comprobar si se puede agregar una negación al nodo actual y en caso de ser así, agregarlo.

Como podemos observar, esta función para realizar esta comprobación específica necesita invocar a métodos de comprobación definidos en el módulo ExtraChecks que ya hemos mencionado. Esto se debe a que los operadores de negación '!' y '~' pueden ser aplicados en muchos lugares, pero no siempre tiene sentido hacerlo. Por ejemplo, centrémonos en la negación '!' que trabaja con los operadores condicionales (pasa igual para '~' solo que con los operadores lógicos).

En Ruby, realizar a = !1 o a = !"hola" está permitido, devolviendo por resultado un false, pero no siempre tiene sentido hacerlo. También permite la doble negación, como la lógica proposicional, y que por tanto un '!!false' devuelve false.

Supongamos que tenemos el siguiente código de entrada que se desea mutar agregando un el operador '!':

```
def relational_and_conditional(a,b,c)
  while !b || a >= c
    if a != 0
      b = fun_b(b)
    end
  end
end
```

En él podría generar varias mutantes, puesto que el operador se puede aplicar en distintos lugares, pero no en todos tiene sentido a pesar de que Ruby lo soportaría. Veamos el primer mutante donde el cambio realizado se marca con un (1º):

```
def relational_and_conditional(a,b,c)
  while !!b (1º) || a >= c
    if a != 0
      b = fun_b(b)
    end
  end
end
```

En este caso la mutación tiene sentido ya que está realizando una doble negación.

La segunda mutación (la marco con un (2º)) y la tercera (la marco con un (3º)) serían:

```
def relational_and_conditional(a,b,c)
  while !b || !a(2º) >= !c(3º)
    if a != 0
      b = fun_b(b)
    end
  end
end
```

¿En este caso que sucede? Tanto a como c tienen pinta de que su objetivo es contener numerales al estar en una operación relacional, pero Ruby permite realizar la negación de un numeral y convertirlo en booleano, aunque un boolean no tenga la operación >=. En este caso no se debería aplicar, pero por ejemplo si en lugar de ser el relacional >= hubiera sido el ==, entonces si se debería de poder aplicar la negación, ya que sí que es posible hacer !true == false.

El caso de == es el mismo al de la mutación 4 donde el (if a!= 0) pasaría a ser (if !a != 0).

Finalmente estaría la mutación 5, en la cual se indica el cambio realizado con (5º):

```
def relational_and_conditional(a,b,c)
  while !b || !a >= !c
    if a != 0
      b = fun_b(!b)(5º)
    end
  end
end
```

En este caso se estaría negando el parámetro que se le pasa a una función(fun_b(!b)). Esto no es correcto sin saber el tipo de dato que contiene y al tratarse de un lenguaje de tipado dinámico esto es imposible, por lo que no se debería de negar aquí y ya se negaría dentro de la función fun_b en caso de que sea posible.

Como podemos observar, hay muchas combinaciones posibles, así que para poder tener todas ellas en cuenta, ha sido necesario agregar el módulo ExtraChecks, que contiene toda la funcionalidad necesaria para poder realizar este tipo de comprobaciones y otras

específicas que ciertos operadores de mutación necesitan como los cambios de signo en los índices, balanceo de asignaciones múltiples y más.

Veamos entonces, en la **Figura 25**, para un código de entrada que mutantes se deberían generar agregando el operador '!'.

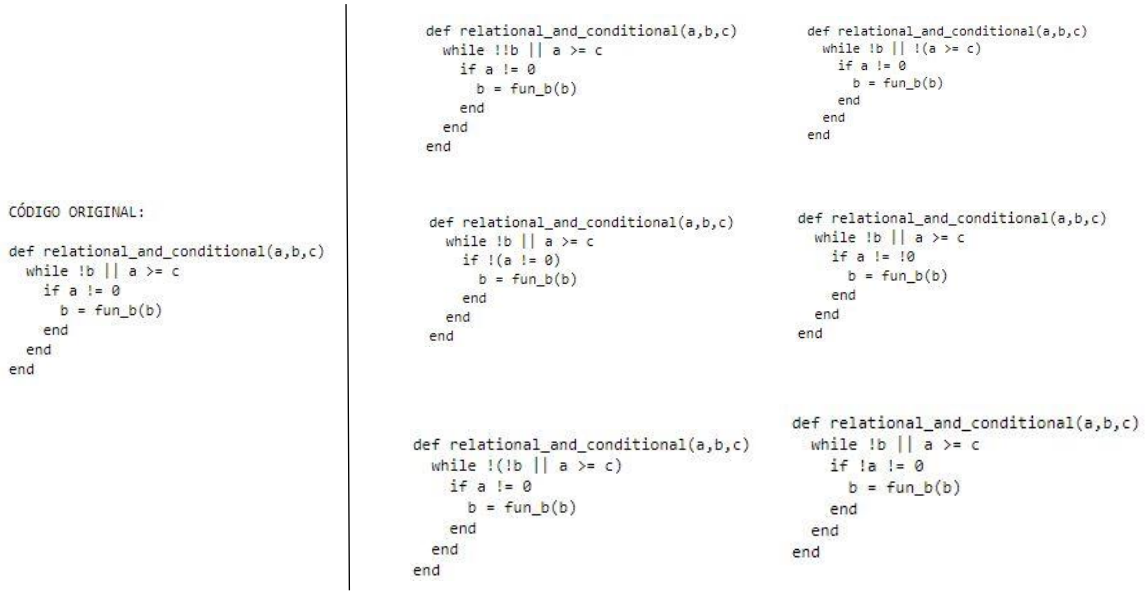


Figura 25 - Código Original y mutantes que se deben generar al agregar !

Para poder generar todas estos mutantes y no generar aquellos que no tienen sentido, se ha tenido que realizar una gran cantidad de filtros y comprobaciones en el módulo ExtraChecks y gracias a ellas la aplicación es capaz de generar mutaciones válidas con lenguajes bien formados para la agregación de operadores de negación, entre otros, sean del tipo que sean.

Una vez que *rewrite()*ha terminado y ha devuelto la cadena con los cambios realizados, se comprueba si esta contiene cambios respecto a la cadena original y en caso de ser así se guardada en el fichero mutationx_y.rb , donde x es el número del operador de mutación que se esté aplicando en ese momento en función al orden en el que fueron seleccionados e y es el número de mutante generado para ese operador. Por ejemplo, si se seleccionó 'sum','res','div' como operadores de mutación en este orden y se genera una segunda mutación para el operador sum, entonces esta se guarda como mutation0_1.rb . Esto luego permite saber a la hora de ejecutar los casos de prueba que archivos coger para ejecutar las pruebas, ya que primero se realizan todas las mutaciones para el primer operador de mutación seleccionado, luego se realizan sus pruebas y después se pasa a realizar todas las mutaciones para el segundo operador de mutación. Cuando se van a realizar las pruebas para estas mutaciones, en el directorio están también los mutantes generados para el operador anterior, así que solo debe coger los del operador actual. Gracias a que en cada interacción se le pasa el número del operador actual que se está aplicando, el programa sabe que ficheros debe coger.

En el caso de que haya modificaciones en el código tras un *rewrite()* también se encarga de recopilar cual ha sido el número de la línea mutada, cual era su contenido original y cual es el nuevo. De este modo se podrá mostrar esta información al presentar los resultados.

```ruby
def check_original_and_new_code(code_mutation, original_code,file_path)
  # Given two codes compare if are the same. If are different, then call to generate
  # a mutation file with the code_mutation. And save the line that has been changed
  # :param code_mutation: the str of the code mutate.
  # :param ast_rewriter: the str of the original code.
  # :param file_path: path of the file with the original content to mutate
  # :return: None
  if code_mutation != original_code
    delimiters = ["\n ", "\n\n", "\n", "\n\n "]
    separator = Regexp.union(delimiters)
    mutate_lines = code_mutation.split(separator).to_set
    original_lines = original_code.split(separator).to_set

    difference_line = (mutate_lines ^ original_lines).to_a
    # difference_line is a Set with the line mutate in pos 1 and original line in pos 0
    # Now check the line number of in the original code. Need to add one because
    # the array start at index 0 but the code file start in line 1, so the element in
    # the index 0 is the line in the row number 1 of the file
    # line_number = original_lines.to_a.find_index(difference_line[0]) + 1
    line_number = FileAndDirectoryManager.get_line_number(file_path, difference_line[0])
    difference_line.append(line_number)
    generate_mutation(code_mutation, difference_line)
  end
end
```

Figura 26 - Códigos para la detección de cambios y recopilación de información de contenido original y contenido nuevo tras la mutación

```ruby
def explore_nodes_and_call_rewriter(code_source_buffer, ast, ast_rewriter, operators, file_path)
  # Loops through the children from the given node and calls the appropriate
  # rewriter to generate the mutation operation based on the type of node.
  # When the rewriter finishes, it checks if something
  # has been mutated with respect to the original code.
  # :param code_source_buffer: A buffer with source code.
  # :param ast: the actual node of the ast to star the loop.
  # :param ast_rewriter: a object of the type RewriteTree to call the correct rewrite
  # :param operators: dict with the actual operator to search and replace
  # :param file_path: path of the file with the original content to mutate
  # :return: None
  ast.children.each { |node|
    if node.is_a?(Parser::AST::Node)
      explore_nodes_and_call_rewriter(code_source_buffer, node, ast_rewriter, operators,file_path)
      code_mutation = ast_rewriter.rewrite(code_source_buffer, node)
      check_original_and_new_code(code_mutation, code_source_buffer.source,file_path)
      # Call checker to evaluate if this type of node has special features that require further changes
      generate_mut_array = ExtraChecks.check_type_of_node(node, ast_rewriter, code_source_buffer, operators)
      generate_mut_array.each { |code_mutation|
        check_original_and_new_code(code_mutation, code_source_buffer.source,file_path)
      }
    end
  }
end
```

Figura 27 - Función para recorrer AST y llamar a rewriter()

En las **Figuras 26 y 27 se** muestran dos funciones implementadas para realizar las tareas mencionadas en este apartado.

4.1.5 Ejecución de pruebas de mutación

Finalmente, tras generar todos los mutantes ya solo queda aplicar los casos de prueba sobre ellos y recopilar los resultados para poder presentárselos al usuario. Como hemos visto, antes de guardar un mutante en un fichero, ya se recopila la información del número de línea que fue mutada en ese mutante y sus contenidos antes y después. Pero falta conocer que mutantes sobreviven a las pruebas proporcionadas y cuáles no.

Los archivos que contienen esta funcionalidad son mutations_and_tests_manager.rb y evaluate_test_results.rb del directorio /manage_test_mutations. El primero se encarga de ejectuar las pruebas para cada mutante generador por cada operador de mutación seleccionado y el segundo se encarga de recopilar los resultados de cada prueba para cada mutante. Debemos recordar que las funciones de mutations_and_tests_manager.rb son llamadas cada vez que se han generado todos los mutantes para un operador de mutación concreto, de este modo si se ha seleccionado 'sum' y 'res', primero se llamará cuando se hayan generado todos los mutantes de 'sum' y posteriormente cuando se hayan generado todos los de 'res'.

Por tanto lo primero que se hace es generar un fichero all.rb que contiene la llamada a todas las pruebas que hubiera en el directorio que el usuario indicara con las mismas, de este modo mediante el comando de terminal 'ruby *local_path*/all.rb' se pueden ejecutar todas las pruebas de golpe sin necesidad de ir una por una. Si es la primera vez que se ejecutan las pruebas para un grupo de operadores de mutación, entonces las pruebas también se ejecutan para el fichero original sin mutaciones y en caso de tener fallos, se corta el proceso y se notifica al usuario. Si el fichero original no tiene errores, entonces se comienzan a realizar las pruebas sobre cada mutante generado. Para ello, ya que las pruebas tienen una invocación del fichero original en su path original, lo que se hace es salvar el código del fichero original temporalmente y sustituir el contenido de este con cada mutación, de este modo el requiere que hay en cada fichero de prueba no tiene que ser modificado y puede ejecutarse sobre el código de cada mutante. La ejecución de las pruebas se hace mediante el comando de terminal:

```
# Execute tests from the generate file 'all.rb' with al the tests
`ruby #{test_file_path} > #{@test_result_path}results_test.txt`.chomp
```

Este permite guardar los resultados obtenidos en el fichero results_test.txt. Tras esto se ejecuta la funcionalidad del fichero evaluate_test_results.rb que se encarga de leer la línea de este fichero que contiene los resultados:

```
6 tests, 6 assertions, 2 failures, 0 errors, 0 pendings, 0 omissions, 0
notifications
```

Tras ello salva cada uno de estos datos de esta línea para cada mutante y de forma general para cada operador de mutación. En el segundo caso, se tiene un registro general para cada operador de mutación seleccionado y en él se van actualizando los datos con cada mutante generada por él que vaya llegando. De este modo podemos tener lo datos genéricos por cada operador de mutación aplicado y para cada mutante.

Una vez que se han generado todos los mutantes de todos los operadores y se han realizado todas las pruebas se devuelve toda la información recopilada para que pueda ser presentada al usuario y se borran los ficheros all.rb y results_test.txt. En caso de que el usuario no haya seleccionado salvar los mutantes, estos también son eliminados y siempre se restaura el contenido original del fichero que se quería mutar antes de finalizar, para que el usuario pueda seguir haciendo uso de él sin alteraciones.

4.2 FrontEnd

Una vez que se ha presentado todo el desarrollo del Backend, vamos a presentar brevemente el desarrollo del FrontEnd. Este se ha estructurado como se puede observar en la **Figura 28**.

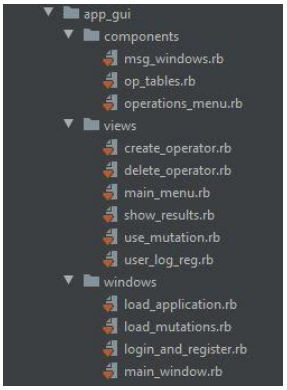


Figura 28 - Estructura GUI

Esta parte ha sido desarrollada ayudándonos de la gema fxruby, la cual nos permite realizar vistas, ventanas y diversos componentes.

El objetivo ha sido tener cuatro ventanas de distintos tamaños para poder mostrar en ellas el contenido de las diferentes vistas de la aplicación, así pues, cada vista contiene los elementos que necesita en cada caso y se van cargando en la ventana que le corresponde según proceda. De este modo no hay que estar creando y eliminando una ventana con cada contenido cada vez que se necesita, sino que se reutilizan las cuatro creadas al principio y de ese modo nos ahorramos tiempo de procesamiento en la generación y destrucción de ventanas [8].

Se ha generado una vista para cada necesidad que ha surgido, por este motivo, existe una para mostrar los campos necesarios para registrarse o acceder a la aplicación con usuario y contraseña o con modo sin usuario. Otra para el menú principal que muestra las diferentes opciones de tareas que se pueden realizar en la aplicación las cuales son: crear un operador, eliminarlo y usar un mutante. Se ha creado otra vista para poder realizar la creación de operadores y en consecuencia una más para poder eliminarlos. Por otro lado, se ha generado una vista para poder realizar las selecciones previas a realizar las mutaciones y obtener los resultados, en la cual se puede especificar: el fichero que se desea mutar, el directorio que contiene los casos de prueba para ejecutar y los operadores de mutación que se quieren aplicar con una descripción de estos. Esta vista también muestra la opción de si se quieren guardar los mutantes generados o no. Finalmente se ha desarrollado una vista para poder mostrar los resultados mediante dos tablas, una con los resultados generales y otra con los específicos. En la segunda, haciendo click sobre el nombre del mutante, si se ha solicitado guardar los mismos, se abrirá su contenido en un pequeño FXText que hay en esta vista para poder ser evaluado.

Algunas señales que se produzcan en los componentes de estas vistas provocan acciones en la aplicación. De este modo cada vez que se produce una señal que implique una funcionalidad, es capturada por un escuchador y se invoca al método correspondiente del Backend para que se encargue de realizarla.

Finalmente se puede observar en la **Figura 28** un directorio /componentes, que contiene aquellos componentes cuya funcionalidad se reutilizan en diferentes vistas de la aplicación. Estos son un menú flotante para poder navegar entre las vistas del menú principal, creación de operadores, eliminación de estos y uso de los mismos, las tabla que presenta los operadores disponibles en la aplicación y su información, que es la misma para la vista de selección de operadores y para la vista de eliminación de los mismos y finalmente un clase que contiene los FXMessageBox de información y error, para que pueda ser invocada desde todas las vistas modificando únicamente el texto de cada Box según corresponda en cada situación.

5 Pruebas y resultados

5.1 Pruebas

El desarrollo de la aplicación se realizó en dos fases, primero se realizó el desarrollo de toda la funcionalidad interna y después se realizó la GUI de usuario. Por este motivo las pruebas realizadas se compusieron de dos partes. Una primera en la que se probó que la funcionalidad interna fuera correcta y otra segunda en la que se comprobó que el funcionamiento fuera correcto al integrar el GUI.

Para comprobar el correcto funcionamiento de la funcionalidad interna, se realizó un main que permitía interactuar con la aplicación a través del terminal, además se generaron doce archivos de ejemplo que contenían las diferentes estructuras que se podían mutar haciendo uso de la aplicación. Se realizó un fichero para comprobar las mutaciones de cambio de operador aritmético y en partes de un string, otro para probar mutaciones de reemplazo en condiciones, otro para probar las mutaciones de reemplazo en bucles, un cuarto con errores de sintaxis para comprobar que la aplicación lo detectaba y notificaba su existencia al usuario, uno más para probar los operadores de mutación de relacionales tanto en condicionales, como en bucles y fuera de estos, otro fichero para probar las operaciones de inserción y borrado, otro para para probar operadores de inserción y borrado de '~', un séptimo para probar el cambio en el acceso al índice de un array, otro fichero para probar el equilibrado de asignaciones múltiples, un noveno para probar la agregación del método freeze, un décimo para probar el cambio de dobles comillas a comillas simples y al contrario, un penúltimo fichero para probar el cambio de 'is_a?' y 'kind_of?' por 'instance_of?' y viceversa y finalmente un último para probar el borrado de returns.

Además, se generaron varias pruebas para probar los distintos casos y detectar si se estaban obteniendo bien los resultados al ejecutar las pruebas sobre el fichero original y los mutantes. De este modo también se programó un conjunto de pruebas que detectaban fallos sobre el código original para comprobar si el programa lo detectaba y lo notificaba al usuario.

En las **Figuras 29, 30 y 31** podemos observar el directorio con los ficheros de ejemplo, algunos contenidos de estos ficheros y un ejemplo de pruebas respectivamente.

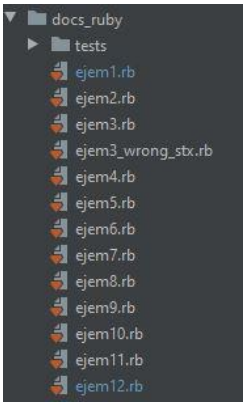
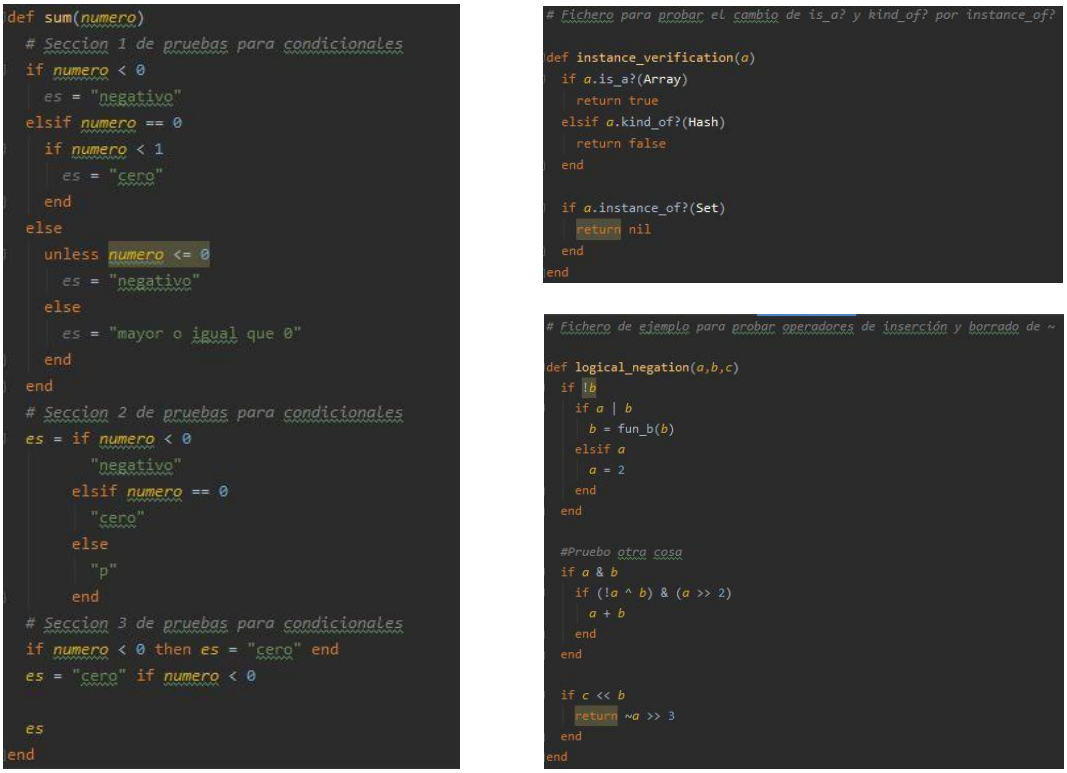


Figura 29 - Contenido directorio ficheros de prueba



```ruby
def sum(numero)
  # Seccion 1 de pruebas para condicionales
  if numero < 0
    es = "negativo"
  elsif numero == 0
    if numero < 1
      es = "cero"
    end
  else
    unless numero <= 0
      es = "negativo"
    else
      es = "mayor o igual que 0"
    end
  end
  # Seccion 2 de pruebas para condicionales
  es = if numero < 0
         "negativo"
       elsif numero == 0
         "cero"
       else
         "p"
       end
  # Seccion 3 de pruebas para condicionales
  if numero < 0 then es = "cero" end
  es = "cero" if numero < 0

  es
end
```

```ruby
# Fichero para probar el cambio de is_a? y kind_of? por instance_of?

def instance_verification(a)
  if a.is_a?(Array)
    return true
  elsif a.kind_of?(Hash)
    return false
  end

  if a.instance_of?(Set)
    return nil
  end
end
```

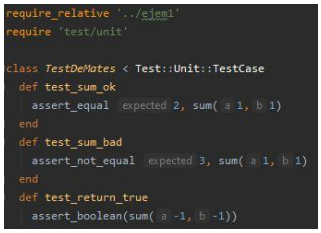
```ruby
# Fichero de ejemplo para probar operadores de inserción y borrado de ~

def logical_negation(a,b,c)
  if !b
    if a | b
      b = fun_b(b)
    elsif a
      a = 2
    end
  end

  #Pruebo otra cosa
  if a & b
    if (!a ^ b) & (a >> 2)
      a + b
    end
  end

  if c << b
    return ~a >> 3
  end
end
```

Figura 30 – Ejemplo contenido ficheros para probar distintos tipos de mutaciones



```ruby
require_relative '../ejem1'
require 'test/unit'

class TestDeMates < Test::Unit::TestCase
  def test_sum_ok
    assert_equal expected: 2, sum( a: 1, b: 1)
  end
  def test_sum_bad
    assert_not_equal expected: 3, sum( a: 1, b: 1)
  end
  def test_return_true
    assert_boolean(sum( a: -1, b: -1))
```

Figura 31 - Ejemplo contenido ficheros de prueba

Una vez que se probó que la funcionalidad interna era correcta, se procedió a desarrollar la GUI y una vez desarrollada se volvieron a utilizar estos ficheros y pruebas para comprobar que todo funcionaba correctamente con la misma y que se obtenían las mutaciones correctas en cada fichero para cada operador de mutación que se pudiera aplicar sobre él y los resultados obtenidos eran los esperados.

Además en la GUI se realizó varias pruebas de usabilidad, que son aquellas que se realizan pidiendo a un usuario que pruebe la aplicación, para comprobar que se detectaban todos los fallos posibles como que un usuario no existiera y se tratara de acceder con él, que se tratara de usar el modo con gestión de usuarios sin tener configurada la base de datos, que se seleccionara un directorio de pruebas que no contuviera pruebas, que se seleccionara un fichero con errores de sintaxis, que se seleccionara un operador de mutación que no realizara cambios en el fichero seleccionado, que se pidiera visualizar el contenido de un mutante cuando no se había solicitado salvarlo, que se intentara generar una nueva operación de mutación con un nombre ya existente para ese usuario y por último que se intentara eliminar una operación de mutación que no hubiese sido generada por el usuario.

5.2 Resultados

Tras finalizar con el desarrollo de la aplicación y haber realizado las pruebas pertinentes, se ha obtenido una aplicación que cumple con los requisitos establecidos y que se corresponde con lo ideado durante la parte de diseño.

De este modo, la aplicación mantiene los mejores aspectos de las aplicaciones ya existentes y mejora aquellos puntos más débiles de estas que habíamos comentado en el **apartado 2.3** de este mismo documento. Por lo que la aplicación finalmente permite trabajar a más de un usuario en el mismo dispositivo local sin necesidad de mezclar los operadores que creen o eliminen unos y otros si se configura la aplicación para poder usar el modo con gestión de usuarios. En caso de que solo un usuario vaya a usar la aplicación en un entorno local, se permite el uso de la aplicación sin haber configurado el modo con gestión de usuarios para facilitar la instalación de aquellos usuarios que no quieran usar este modo y en general, permitiendo al usuario configurar su aplicación según desee. Además cuenta con un total de 29 operadores de mutación diferentes incluidos con la instalación de la aplicación que permiten realizar mutaciones de reemplazo, agregación y eliminación de distintos elementos, pero además, estos son extensibles ya que permite a los usuarios agregar sus propios operadores de mutación del tipo reemplazo, por lo que se pueden agregar nuevos operadores de mutación en un futuro si surgen nuevos operadores aritméticos, lógicos, condicionales, relacionales o de asignación. También permite operadores de reemplazo para poder sustituir llamadas a funciones, nombres de variables, o contenido de cadenas tanto sin con dobles comillas o con comillas simples. Por otro lado, la aplicación permite a los usuarios eliminar todos aquellos operadores de mutación creados por ellos mismos que ya no vayan a ser usados o hayan quedado desfasados. Además se encarga de gestionar correctamente todos los errores que pudieran surgir durante una ejecución, notificando al usuario cada uno de ellos con su respectivo mensaje, de este modo detecta aquellas ocasiones en la que un usuario intenta acceder con el modo de gestión de usuarios cuando este no ha sido configurado, detecta cuando se intenta crear un operador con el mismo nombre que uno ya existente, es capaz de reconoces si el operador que se desea eliminar ha sido generado por el usuario que intenta borrarlo o no, también detecta si el directorio que el usuario indica como el que contiene las pruebas se trata de un directorio con pruebas o no, también comprueba si el fichero original pasa las pruebas que se han indicado y mira si el fichero a mutar contiene errores de sintaxis o si es un fichero vacío. Finalmente es capaz de detectar si un operador de mutación no ha generado mutantes y si se ha seleccionado la opción de guardar mutantes, se notifica al usuario la ruta en la que se están guardando.

Además, los resultados obtenidos se presentan en dos grupos, por un lado, los generales para cada operador de mutación aplicado y por otro para cada mutante generado por separado. Por lo que el usuario puede tener una vista general y una más especifica si así lo desea. En cuanto a los resultados generales para cada operador de mutación se muestran la cantidad de mutantes generados, el número de estos que ha sobrevivido y el número que ha fallecido junto con el porcentaje de mutación, por lo que el usuario puede ver la calidad de sus pruebas para cada operador de mutación, siendo alta si el porcentaje de mutación es alto y siendo baja si el porcentaje de mutación es bajo. Además, puede comprobar uno a uno los resultados de cada mutante generado, pudiendo ver si ha sobrevivido o no, que operador de mutación lo ha generado, que número de línea se ha mutado, que contenía dicha línea originalmente y cual es su nuevo contenido tras la mutación, el número de pruebas a las que ha sido sometida y cuantas de estas pruebas han dado error, cuantas fallo y cuantas han sido superadas con éxito. Por lo que la cantidad de resultados mostrados permiten al usuario poder conocer con

exactitud la calidad de sus pruebas y cuales le faltan por implementar para mejorar la calidad de estas en función a lo que observe en los datos de cada mutante generado.

Además, la aplicación cuenta con un interfaz gráfico de usuario que permite hacer uso de esta de una forma mucho más sencilla y que evita obligar al usuario a tener que aprender comandos para poder usarla, escribir rutas completas para indicar ficheros o directorios, buscar los resultados entre párrafos interminables en un terminal, etc. De este modo se presenta toda la funcionalidad de una forma sencilla y amigable y permite al usuario poder sacar el máximo partido a la aplicación con rapidez. Esta interfaz ayuda al usuario en la búsqueda de ficheros y directorios a la hora de establecer los datos para realizar las mutaciones y le muestra los resultados de una forma limpia y ordenada, por lo que su análisis es mucho más sencillo y efectivo. También permite al usuario abrir los propios mutantes generados, si estos han sido guardados, en la propia aplicación en la misma vista en la que se muestran los resultados haciendo click sobre el nombre del mutante que desea abrir en la tabla que muestra los resultados específicos para cada uno de ellos. De este modo el usuario podrá visualizar el mutante generado al mismo tiempo que puede observar los resultados, lo que le permite obtener mejores conclusiones y realizar mejores análisis de los posibles fallos que sus pruebas pueden no haber cubierto.

Finalmente, la aplicación permite el almacenamiento seguro de datos de riesgo en su base de datos, por lo que es una aplicación segura en el modo con gestión de usuarios ante el robo de esta.

A continuación, se procede a mostrar algunas imágenes del estado final de la aplicación con el GUI.

Las **Figuras 32, 33, 34, 35, 36, 37, 38 y 39** muestran el resultado final de la aplicación desarrollada.

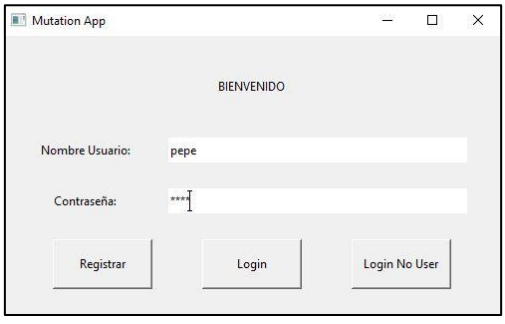


Figura 32 – Vista acceso

Figura 33 – Vista menú principal y menú flotante

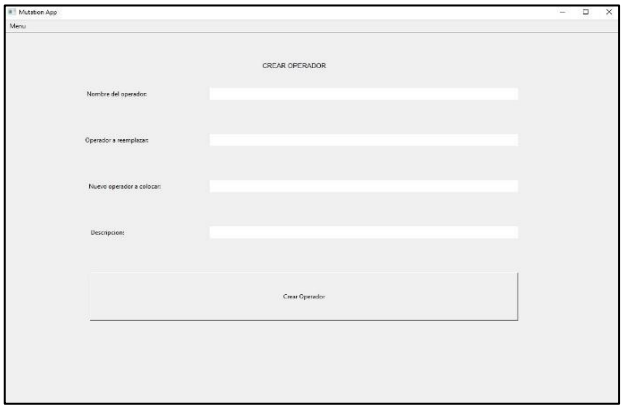


Figura 34 – Vista para crear operadores de mutación

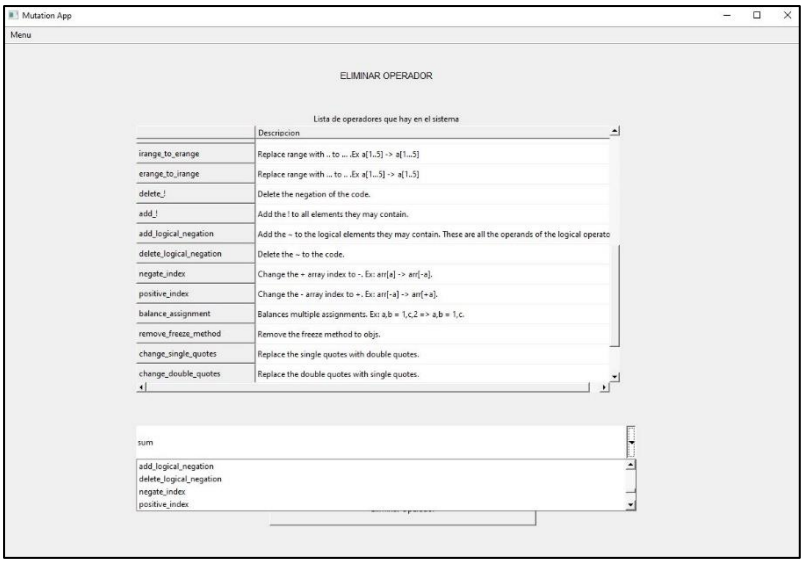


Figura 35 – Vista para eliminación de operadores de mutación

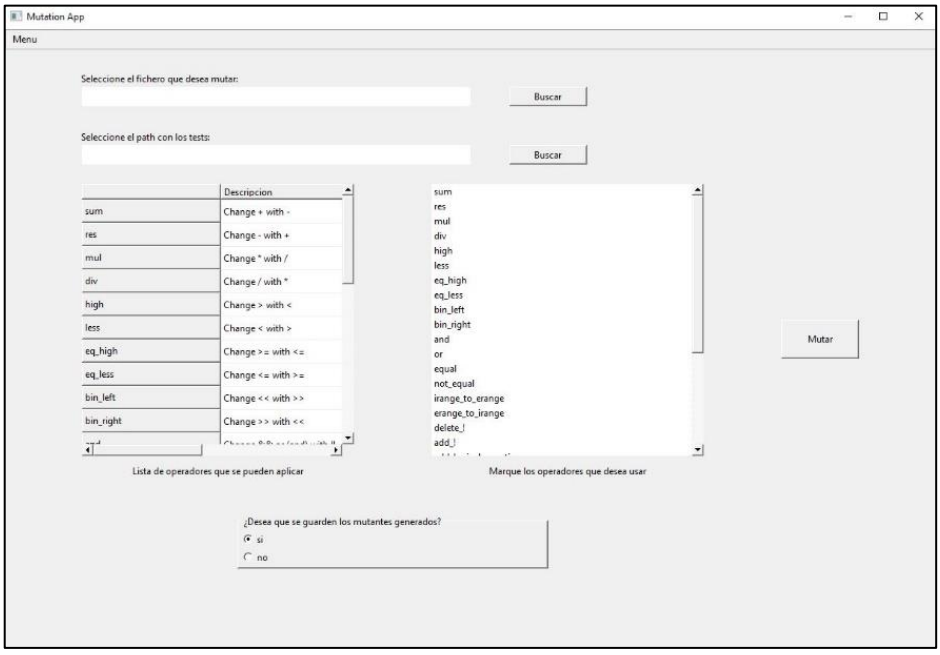


Figura 36 - Selección de operadores a aplicar, fichero a mutar y pruebas a ejecutar

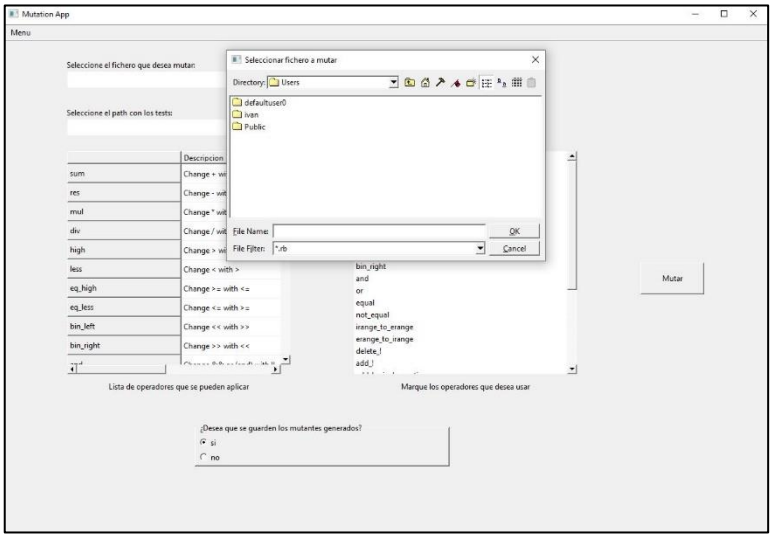


Figura 37 – Búsqueda de ficheros de forma visual

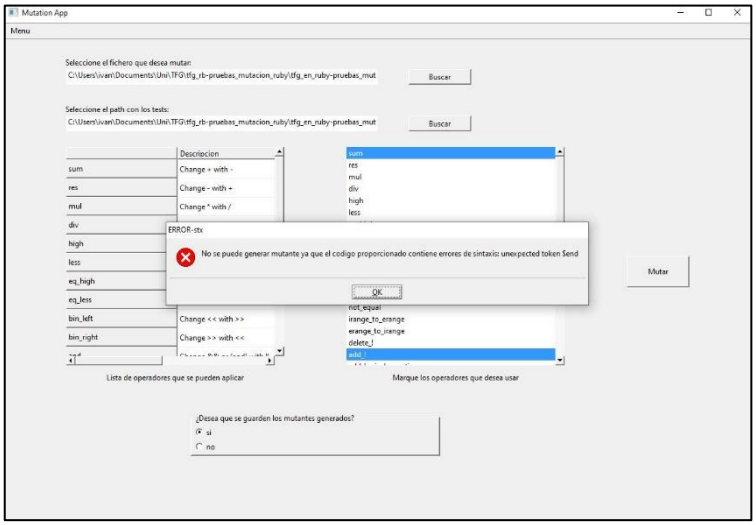


Figura 38 – Ventana de notificación error de sintaxis en el fichero seleccionado

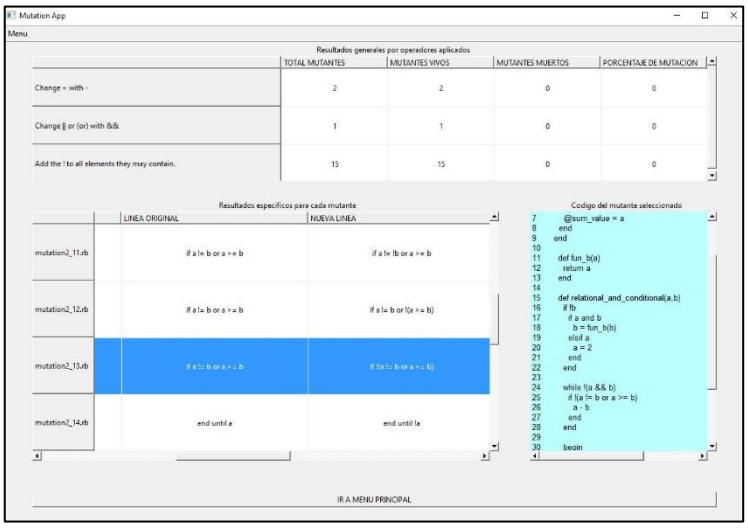


Figura 39 – Vista para mostrar resultados

6 Conclusiones y trabajo futuro

6.1 Conclusiones

Como se ha comentado a lo largo de este documento, el desarrollo de pruebas es fundamental para comprobar la calidad del código desarrollado y el correcto funcionamiento de este y por eso es una de las fases más importantes en cualquier proyecto software. Estas pruebas permiten al desarrollador tener confianza absoluta en el código que ha desarrollado y a la hora de programar nuevas funcionalidades o realizar ampliaciones de la aplicación, le permiten tener la certeza de que lo que estaba hecho hasta ese momento no tenía ningún error.

Pero desarrollar pruebas que sean de calidad y que comprueben todos los aspectos necesarios de la aplicación es todo un arte y solo los desarrolladores más experimentados son capaces de realizar estas pruebas de la mejor calidad posible, pero aun así no siempre lo consiguen. Por eso es muy importante tener algún medio que permita comprobar la calidad de las pruebas implementadas, para que los nuevos desarrolladores comiencen a adquirir experiencia en el desarrollo de pruebas y para que los más experimentados afiancen y mejoren sus conocimientos con nuevos casos que en ocasiones son de muy difícil visión para el ojo humano.

El modo de comprobar la calidad de las pruebas es mediante el uso de pruebas de mutación, las cuales son muy eficaces para este fin, pero son de un alto costo de desarrollo si se deben realizar manualmente. Por este motivo es imprescindible la existencia de herramientas que realicen estas tareas de formas automatizada y permitan a los usuarios medir la calidad de sus pruebas y mejorar las mismas o introducir nuevas si así lo requieren de una forma sencilla y rápida.

Es por todo esto que la aplicación desarrollada en este TFG se considera de gran importancia para dar soporte a los desarrolladores de código Ruby y facilitar su trabajo, dotándoles de una aplicación que les permite realizar pruebas de una forma fácil e intuitiva y mejorando las herramientas ya existentes para este lenguaje que desempeñan tareas similares, permitiendo al usuario mayor flexibilidad de uso en la misma.

Por lo que se considera que una funcionalidad básica para medir la calidad de las pruebas desarrolladas y ayudar a los programadores a mejorar sus códigos queda cubierta con esta aplicación.

6.2 Trabajo futuro

Como trabajo futuro se plantea la posibilidad de agregar nuevos operadores de mutación por defecto en la aplicación para que el abanico de posibilidades sea más amplio.

También se plantea la posibilidad de migrar los operadores de mutación a una base de datos, ya sea relacional o no, ya que puede ser orientada a documentos como MongoDB , con el objetivo de que si en un futuro los datos en la aplicación crecen considerablemente, bien por la existencia de nuevos operadores o por la ampliación de la aplicación a nuevos lenguajes, los tiempos de búsqueda, carga y escritura de operadores sean más rápidos y eficaces.

Referencias

[1] Crude-Mutant : https://kellysutton.com/2018/12/28/better-code-through-mutation-testing.html [Accedido: Marzo 2020]

[2] Mutant: https://neontapir.github.io/professional/2017/03/20/mutation-testing-in-ruby/[Accedido: Marzo 2020]

[3] Mutest: https://cognitohq.com/how-to-write-better-code-using-mutation-testing/[Accedido: Marzo 2020]

[4] Mutest Change_Log: https://github.com/dgollahon/mutest/blob/master/CHANGELOG.md[Accedido: Marzo 2020]

[5] Ruby: http://es.tldp.org/Manuales-LuCAS/doc-guia-usuario-ruby/guia-usuario-ruby.pdf[Accedido: Marzo 2020]

[6] Parser: https://github.com/whitequark/parser[Accedido: Marzo 2020]

[7] FxRuby: https://github.com/larskanis/fxruby[Accedido: Mayo 2020]

[8] FxRuby manual: http://index-of.es/Ruby/FXRuby%20-%20Create%20Lean%20and%20Mean%20GUIs%20with%20Ruby%20(2008).pdf [Accedido: Mayo 2020]

[9] Árbol de búsqueda: https://www.wikiwand.com/es/%C3%81rbol_binario_de_b%C3%BAsqueda[Accedido: Abril 2020]

[10] Bcrypt: https://solidgeargroup.com/password-nodejs-mongodb-bcrypt/[Accedido: Mayo 2020]

[11] Bcrypt documentation: https://github.com/codahale/bcrypt-ruby[Accedido: Mayo 2020]

[12] Programming Ruby 1.9 & 2.0 (4th Edition). The Pragmatic Programmers' Guide. Dave Thomas. The Pragmatic Programmers (2013).

[13] Ranking Lenguajes más usados https://cipsa.net/lenguajes-programacion-mas-populares-2019-2020-ranking-tiobe/[Accedido: Mayo 2020]

[14] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. IEEE Computer, 11(4):34-41, 1978.

[15] PostgreSQL : https://www.postgresql.org/about/ [Accedido: Marzo 2020]

[16] Y.-S. Ma, J. Offutt, Y. R. Kwon. MuJava: a mutation system for java. ICSE 2006: 827-830.

[17] N. Li, M. West, A. Escalona, V. H. S. Durelli. Mutation testing in practice using Ruby. ICST Workshops 2015: 1-6.

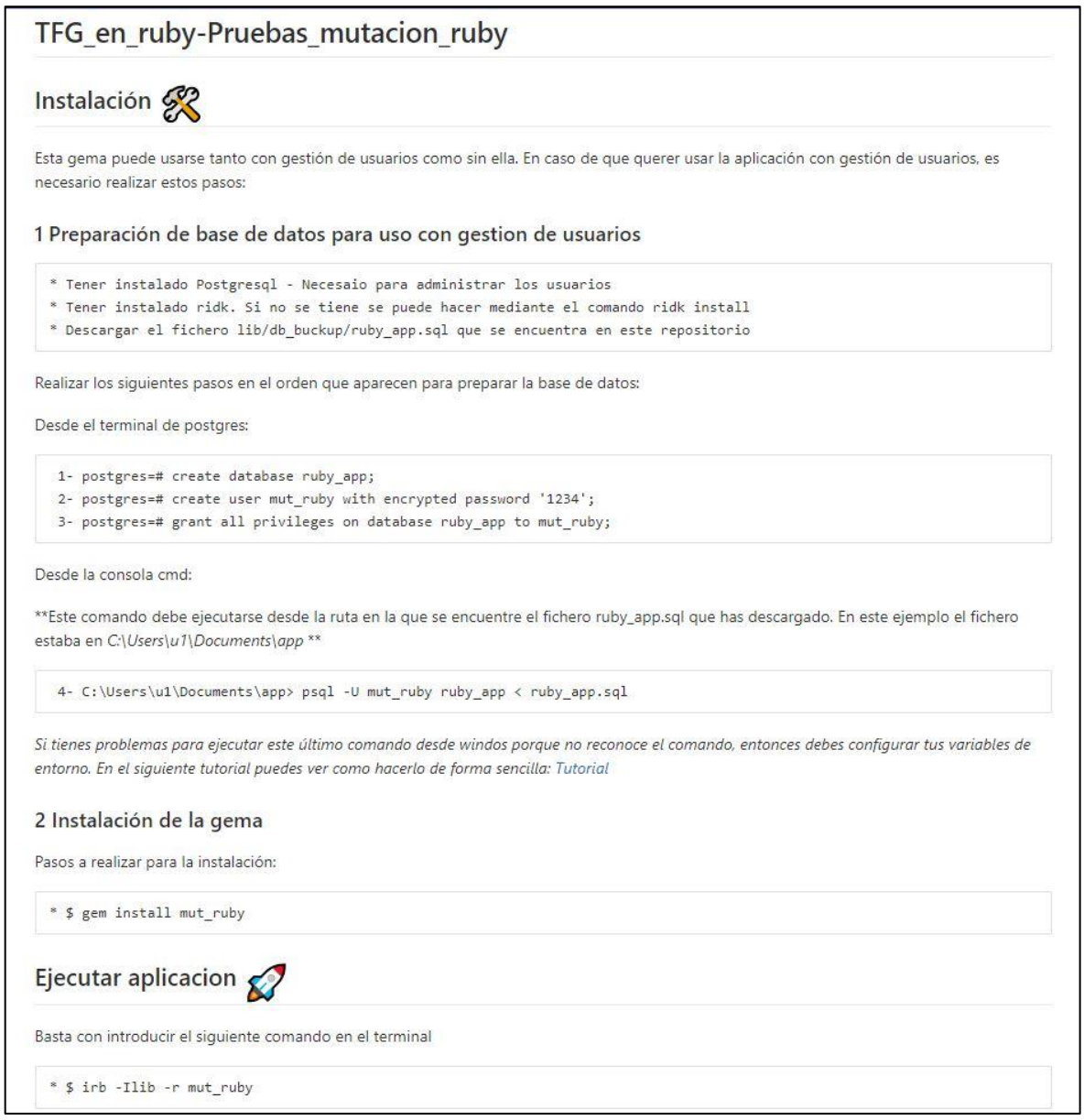
Glosario

API	Interfaz de Programación de Aplicaciones
COMPILAR	Convertir un programa en lenguaje máquina a partir de otro programa de computadora escrito en otro lenguaje.
GUI	Interfaz gráfica de usuario.
INTERPRETE	Software que recive un programa en alto nivel, lo analiza y lo traduce a codigo máquina.
GEMA	En Ruby, son paquetes de librerías que se instalan en el Sistema y pueden ser usadas durante el desarrollo de un programa.
GIT	Software de control de versiones.
SALT	Cadena aleatoria utilizada en criptografía para el salvado de claves forma segura en base de datos.
HASH	Funciones usadas en criptografía para cifrar una entrada.
PK	Primary key de una base de datos relacional que permite establecer el identificador único de cada elemento de a base de datos.
QUERY	Consulta a una base de datos que permite obtener elementos de la misma, insertar nuevos y eliminarlos.
PATH	Referencia o ruta a un archivo o directorio del sistema.
VISTA	En una GUI, elemento que almacena el contenido de una ventana y que permite reutilizarlas.
OPERADOR DE MUTACIÓN	Se trata del cambio que se quiere aplicar a un código para generar una mutación.

Anexos

A Manual de instalación

El manual de instalación podrá encontrarlo cualquier usuario en el Readme.md del repositorio. A continuación, en la **Figura 40** se muestra este fichero.



TFG_en_ruby-Pruebas_mutacion_ruby

Instalación

Esta gema puede usarse tanto con gestión de usuarios como sin ella. En caso de que querer usar la aplicación con gestión de usuarios, es necesario realizar estos pasos:

1 Preparación de base de datos para uso con gestion de usuarios

```
* Tener instalado Postgresql - Necesaio para administrar los usuarios
* Tener instalado ridk. Si no se tiene se puede hacer mediante el comando ridk install
* Descargar el fichero lib/db_buckup/ruby_app.sql que se encuentra en este repositorio
```

Realizar los siguientes pasos en el orden que aparecen para preparar la base de datos:

Desde el terminal de postgres:

```
1- postgres=# create database ruby_app;
2- postgres=# create user mut_ruby with encrypted password '1234';
3- postgres=# grant all privileges on database ruby_app to mut_ruby;
```

Desde la consola cmd:

**Este comando debe ejecutarse desde la ruta en la que se encuentre el fichero ruby_app.sql que has descargado. En este ejemplo el fichero estaba en *C:\Users\u1\Documents\app* **

```
4- C:\Users\u1\Documents\app> psql -U mut_ruby ruby_app < ruby_app.sql
```

Si tienes problemas para ejecutar este último comando desde windos porque no reconoce el comando, entonces debes configurar tus variables de entorno. En el siguiente tutorial puedes ver como hacerlo de forma sencilla: Tutorial

2 Instalación de la gema

Pasos a realizar para la instalación:

```
* $ gem install mut_ruby
```

Ejecutar aplicacion

Basta con introducir el siguiente comando en el terminal

```
* $ irb -Ilib -r mut_ruby
```

Figura 40 - Manual de instalación y configuración

En el enlace https://gitlab.com/ivan_diaz_moreno/tfg_en_ruby-pruebas_mutacion_ruby/-/tree/feature/app_gui_users se puede encontrar el repositorio con todo el código de la aplicación desarrollada. La rama *feature/app_gui_users* es la que contiene la última versión.

B Operadores por defecto del sistema

Lista de operadores que vienen por defecto con la instalación de la gema y que se encuentran en la aplicación para ser usados.

NOMBRE	DESCRIPCIÓN
sum	Change + with – and += with -=
res	Change - with + and -= with +=
mul	Change * with / and *= with /=
div	Change / with * and /= with *=
high	Change > with <
less	Change < with >
eq_high	Change >= with <=
eq_less	Change <= with >=
bin_left	Change << with >>
bin_right	Change >> with <<
and	Change && or 'and' with ||
or	Change || or 'or' with &&
equal	Change == with !=
not_equal	Change != with ==
irange_to_erange	Replace range with .. to ... .Ex a[1..5] -> a[1...5]
erange_to_irange	Replace range with ... to .. .Ex a[1...5] -> a[1..5]
delete_!	Delete the negations of the code.
add_!	Add the ! to all elements they may contain.
add_logical_negation	Add the ~ to the logical elements they may contain. These are all the operands of the logical operators.
delete_logical_negation	Delete the ~ to the code.
negate_index	Change the + array index to -. Ex: arr[a] -> arr[-a].
positive_index	Change the - array index to +. Ex: arr[-a] -> arr[+a].
balance_assignment	Balances multiple assignments. Ex: a,b = 1,c,2 => a,b = 1,c.
remove_freeze_method	Remove the freeze method to objs.
change_single_quotes	Replace the single quotes with double quotes.
change_double_quotes	Replace the double quotes with single quotes.
change_simple_class_check	Replace instance_of? calls with is_a? call.
delete_returns	Delete return word from returnsm
change_exhaustive_class_check	Replace is_a? or kind_of? calls with instance_of? call.

Tabla 1 - Operadores de mutación por defecto del sistema

C Ejemplo de ejecución con creación, uso y eliminación de operador

Este anexo presenta un ejemplo de ejecución ilustrado, que permite ver el proceso para crear, usar y eliminar un operador. En las Figuras que se pueden observar, una elipse azul indica el botón seleccionado que permite cambiar de vista y que en este caso muestra el paso de una Figura a la siguiente.

En primer lugar, hay que acceder a la aplicación, en este ejemplo el modo con gestión de usuarios está configurado para ser usado, por lo que aprovecharemos este hecho para realizar todas las acciones como un usuario registrado.

En la **Figuras 41 y 42** se puede observar la creación de un nuevo usuario y su acceso a la aplicación. Cabe destacar que si el usuario se está registrando, si el registro ha sido satisfactorio, accederá directamente a la aplicación sin necesidad de tener que realizar el login tras el registro.

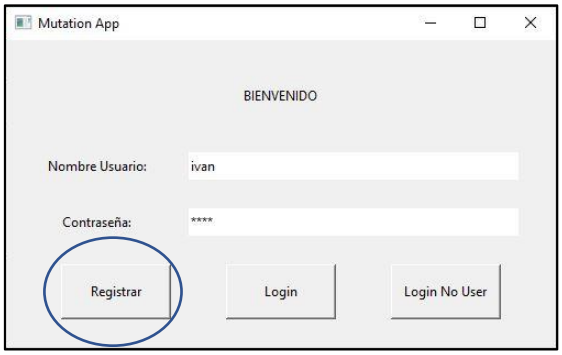


Figura 41 – Vista inicio de sesión

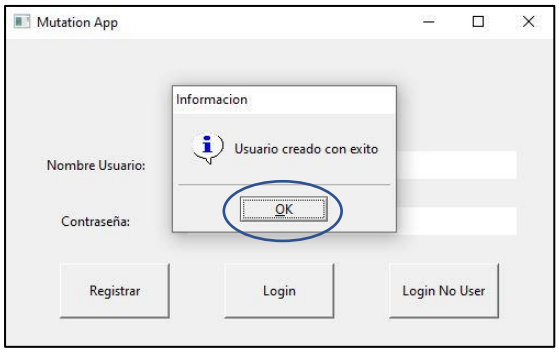


Figura 42 – Confirmación registro

Una vez que el usuario ha sido creado con éxito, el usuario se encontrará en el menú principal que se puede observar en la **Figura 43**.

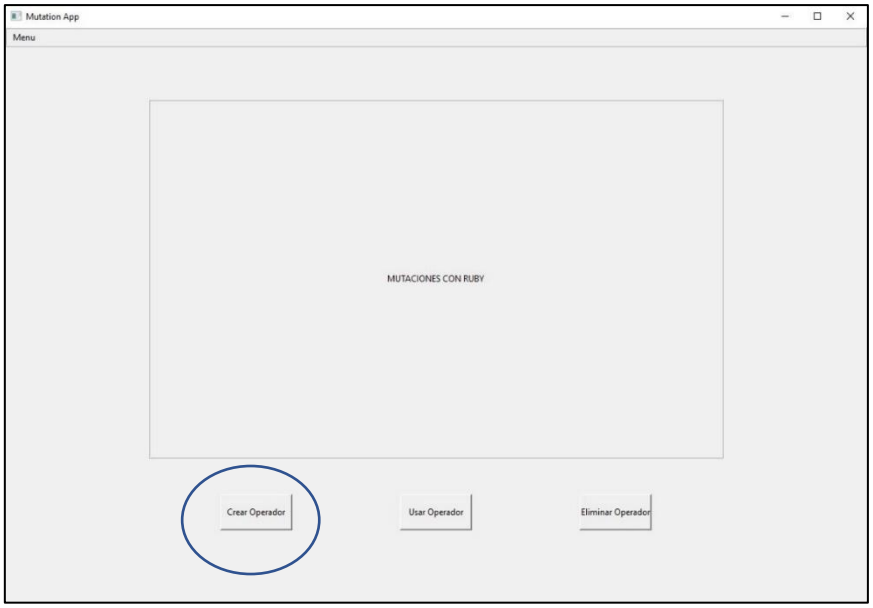


Figura 43 - Menú principal

Desde el menú principal se pueden acceder a todas las funcionalidades de la aplicación a excepción del mostrado de resultados, que solo es accesible tras haber accedido a usar un operador y haber generado mutantes.

Haciendo click sobre el botón Crear Operador accedemos a la vista que podemos observar en la **Figura 44**, la cual permite al usuario generar nuevos operadores para almacenarlos en la aplicación y así poder usarlos cuando él desee. En esta **Figura 44** se muestran los campos completos con la información asociada al operador que estamos generando. En caso de que no todos los campos estuvieran completos, el nombre ya existiera o alguno de los campos contuviera caracteres latinos o comillas, la aplicación mostraría un mensaje de error especificando el problema en cada caso y no dejaría crear el operador hasta que todos los problemas fueran corregidos.

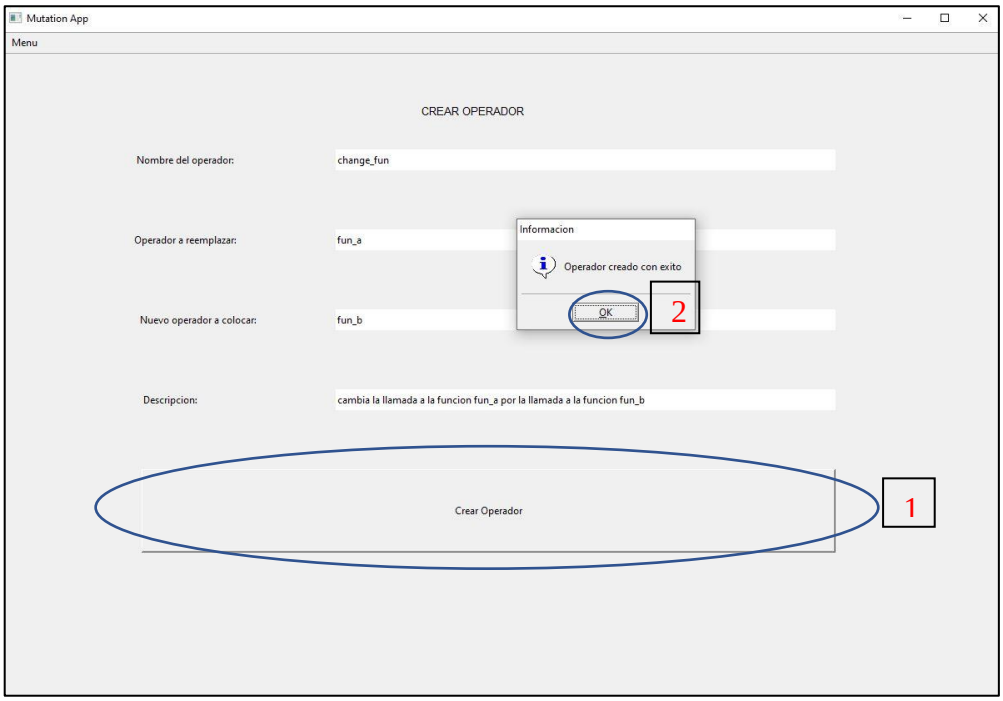


Figura 44 – Confirmación de creación de operador de mutación correcta

Una vez que los campos han sido completados correctamente y tras pulsar en el botón marcado con 1 en la **Figura 44** aparece la ventana flotante de información para indicar que el operador has sido creado correctamente. Tras pulsar en el botón marcado con 2 en la **Figura 44**, se vuelve a la vista que se observa en la **Figura 45**.

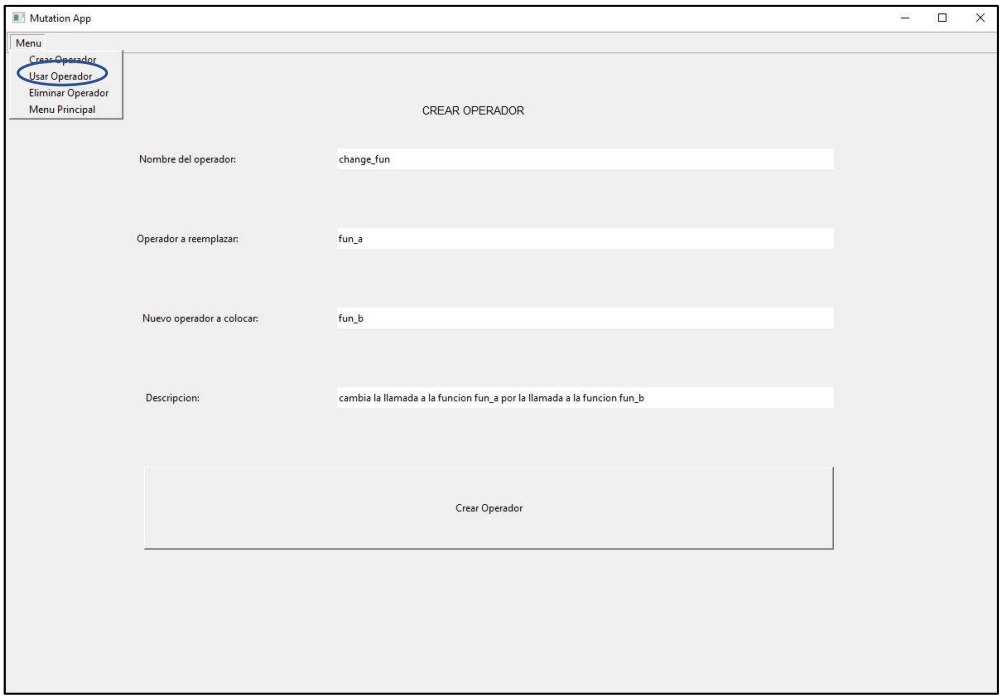


Figura 45 – Vista menú flotante desde vista para creación de operadores.

Usaremos el menú flotante que hay en ella para acceder a la vista de usar un operador y poder usar el operador que acabamos de crear.

En la **Figuras 46 y 47** se muestra el código que se quiere mutar y las pruebas que hay implementadas para dicho código.

```
def fun_a(a,b)
  c = b + a
  if c > 5
    return c + 1
  else
    return c
  end
end

def fun_b(a,b)
  c = a + b
  if c > 5
    return c + 2
  else
    return c
  end
end

def start(x,y)
  fun_a(x,y)
end
```

Figura 46 – Código de fichero a mutar

```
require_relative '../ejem1'
require 'test/unit'

class TestDeMates < Test::Unit::TestCase
  def test_sum_ok
    assert_equal  expected 2, start( x 1, y 1)
  end
  def test_sum_bad
    assert_equal  expected 8, start( x 6, y 1)
  end
end
```

Figura 47 – Conjunto de pruebas proporcionadas

Conociendo los ficheros que vamos a proporcionar a la aplicación, completamos los campos de la vista de usar un operador como se muestra en la **Figura 48**. En este caso dejamos marcada la opción guardar operadores y seleccionamos el operador que acabamos de crear

con nombre change_fun de entre todos los posibles. En caso de que el campo con el fichero a mutar o el campo para conocer el path de los tests no haya sido modificado, se notificará mediante un mensaje de error que estos campos deben ser completados. En caso de que no se haya seleccionado ningún operador de mutación para aplicar, ocurrirá lo mismo. Por otro lado, si el fichero que se desea mutar está vacío o contiene errores de sintaxis, la aplicación también se lo comunicará al usuario mediante otra ventana de error y finalmente si se ha especificado un path con las pruebas que no contiene pruebas, entonces también se le hará saber al usuario mediante su correspondiente mensaje. En este ejemplo todo se ha hecho correctamente a la primera por lo que ningún mensaje ha sido mostrado.

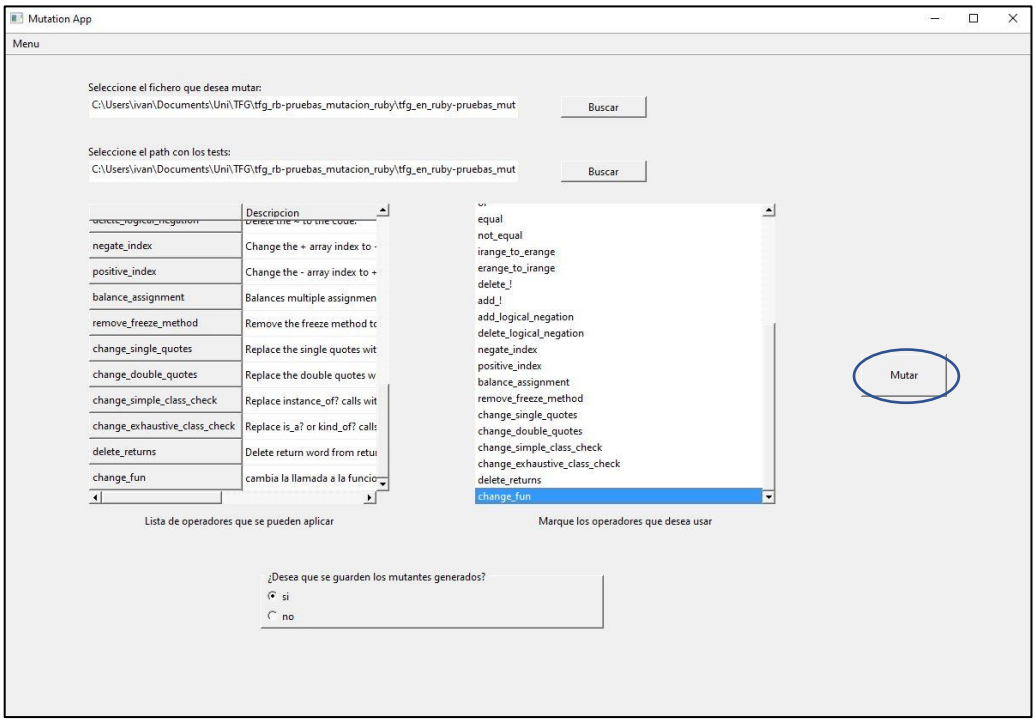


Figura 48 – Vista para configurar mutaciones a realizar

Tras hacer click en el botón marcado de la **Figura 48**, como se ha solicitado que se guarden los mutantes generados, se mostrará un mensaje indicando en que ruta se guardarán estos mutantes y se notificará con una pequeña ventana que los mutantes se están generado y que las pruebas están siendo pasadas para que el usuario no piense que la aplicación se ha quedado colgada. En caso de que el fichero original a mutar no pasé las pruebas especificadas o no se hayan generados mutantes para él con los operadores seleccionado, el usuario lo sabrá inmediatamente gracias a la información que se le mostrará por medio de una ventana flotante y en dicho caso se mantendrá en la ventana actual que se observa en la **Figura 48**. Mientras que si todo ha ido bien se pasará a la ventana que muestra los resultados. Esta se puede observar en la **Figura 49**.

En caso de que se seleccione el nombre de un mutante generado, si se ha solicitado que fuera guardado, se mostrará el contenido de este en el recuadro marcado en azul.

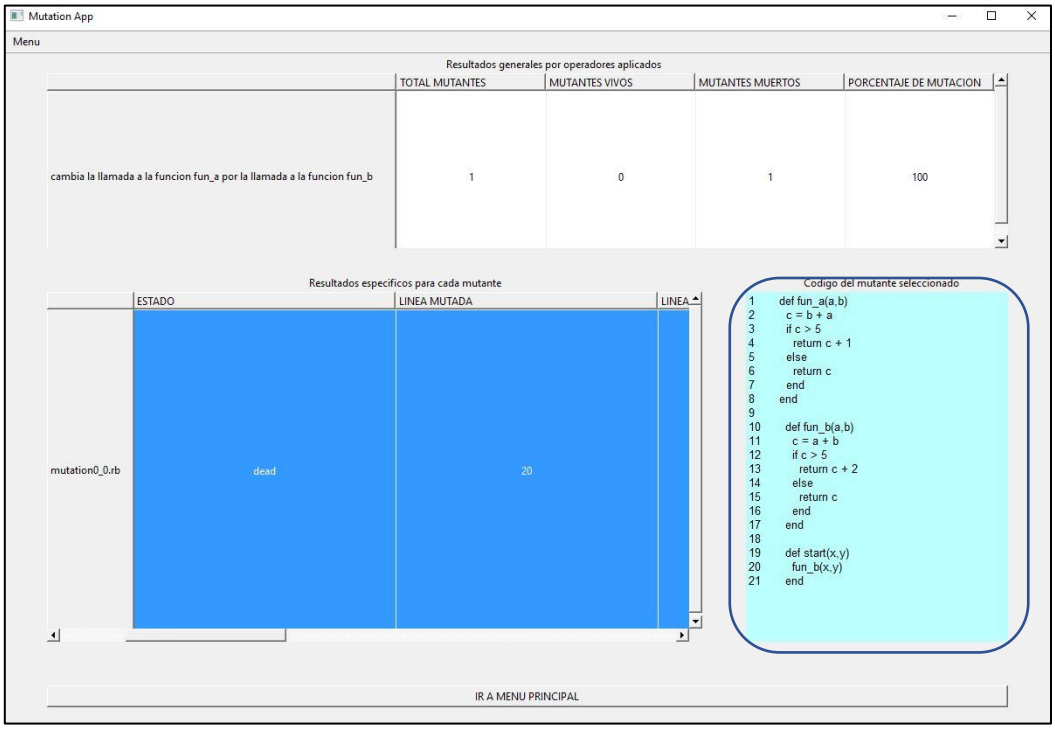


Figura 49 – Vista para mostrar resultados obtenidos

Como podemos observar en la **Figura 49**, se ha generado un mutante que no ha sobrevivido a las pruebas proporcionadas, por lo que podemos saber que nuestras pruebas si cubren este caso y que por tanto nuestro porcentaje de mutación es del 100% para este operador de mutación en este fichero. Además en la sección marcada en azul celeste podemos observar el fichero mutante generado y en la línea 20 se puede ver que la antigua llamada a *fun_a()* ha sido cambiada por *fun_b()*. Para ver un ejemplo más detallado de todos los datos mostrados en esta vista, se recomienda ver el **Anexo D**, donde se muestra un caso más detallado solo de resultados.

Finalmente, desde la vista que se observa en la **Figura 49**, haciendo uso del menú flotante, se puede acceder directamente a la vista para eliminar operadores. Esta puede ser observada en la **Figura 50**.

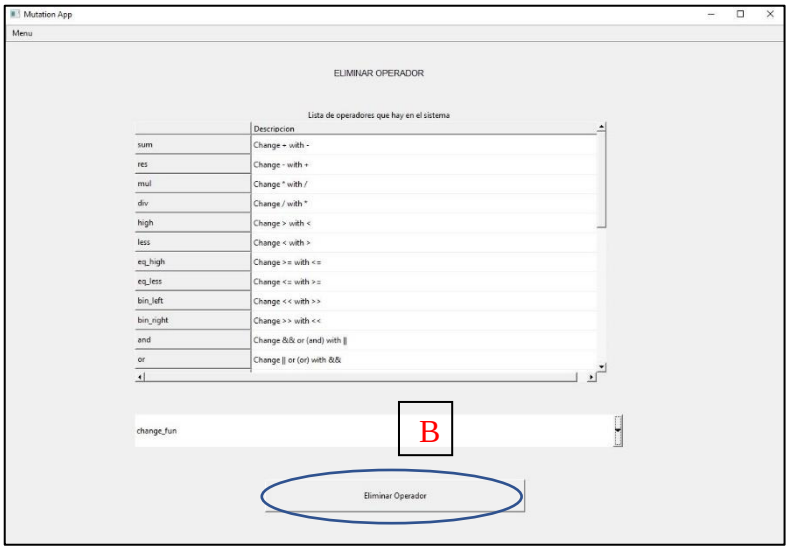


Figura 50 – Vista eliminación de operadores de mutación

En esta vista el usuario puede ver todos los operadores que hay en el sistema para él y en el comboBox marcado con una B en la **Figura 50**, puede seleccionar aquel que quiere eliminar. Tras esto basta con pulsar el botón Eliminar Operador, marcado con una elipse azul en la **Figura 50**. En caso de que el operador que se intenta borrar no haya sido creado por el usuario, entonces se le notificará que no es posible y se mantendrá en la misma vista. Mientras que, si todo es correcto, se eliminará el operador y se mostrará mediante una venta de información de que el operador ha sido eliminado correctamente.

De este modo el usuario si vuelve acceder a la vista de usar un operador de la **Figura 48**, ya no encontrará el operador para ser usado.

D Resumen resultados mostrados por la aplicación

En este apartado D del anexo se proceden a presentar de una forma más visual los resultados que se pueden obtener tras una ejecución.

Por ejemplo, supongamos que tenemos el siguiente fichero de entrada que se muestra en la **Figura 51**.

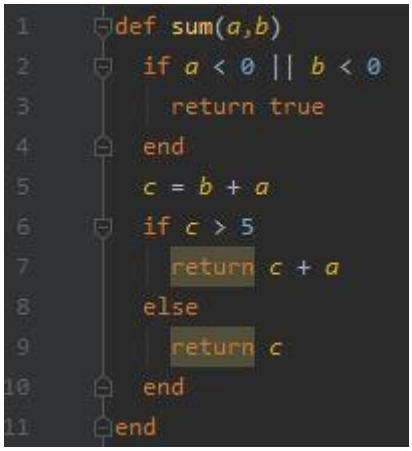


Figura 51 – Contenido fichero a mutar

Y las pruebas facilitadas para este fichero son las que siguen en las **Figuras 52 y 53**:

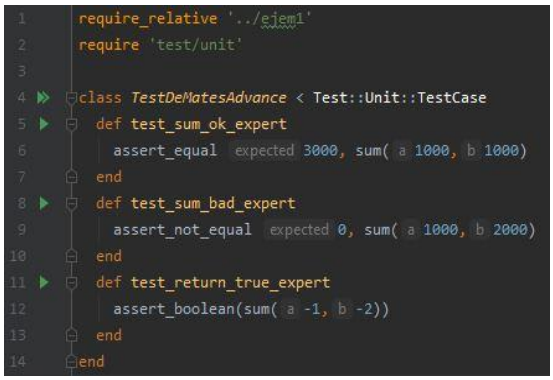


Figura 52 – Pruebas proporcionadas

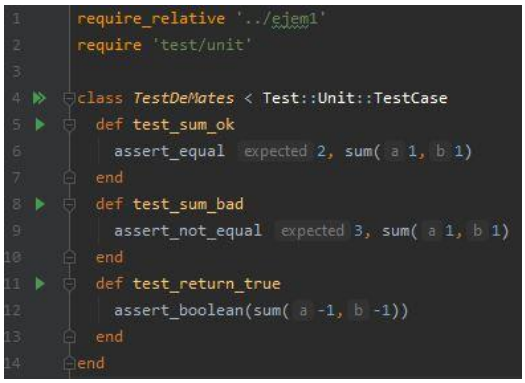


Figura 53 – Pruebas proporcionadas

Si seleccionan los operadores que vienen por defecto en el sistema ‘sum’, ‘or’ y ‘high’ como se ve en la **Figura 54**, se obtienen los resultados genéricos de la **Figura 55** y los específicos de las **Figuras 56,57,58,59,60**.

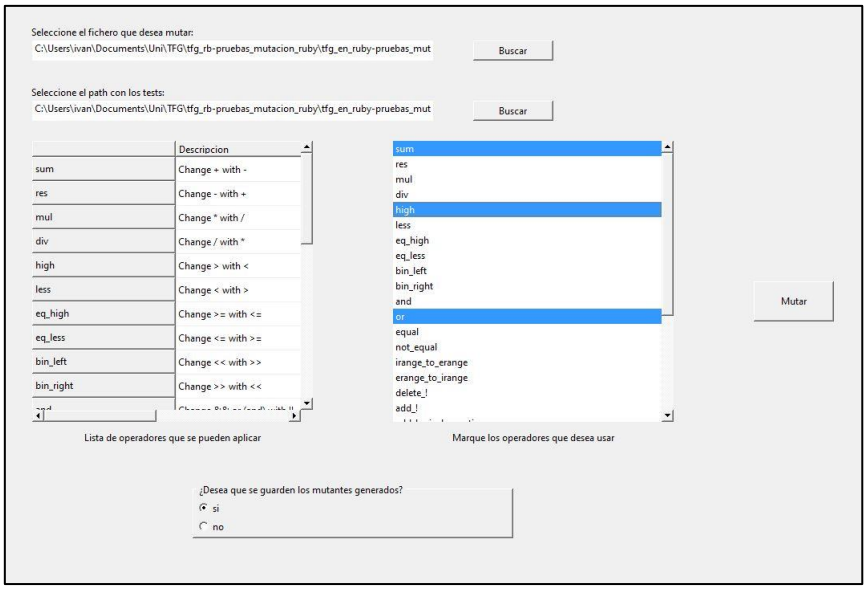


Figura 54 – Vista para realizar configuración de mutaciones

En la **Figura 55** que se observa a continuación podemos ver los resultados genéricos obtenidos para cada operador de mutación.

	TOTAL MUTANTES	MUTANTES VIVOS	MUTANTES MUERTOS	PORCENTAJE DE MUTACION
Change + with -	2	0	2	100
Change > with <	1	0	1	100
Change || or (or) with &&	1	1	0	0

Figura 55 – Resultados por operador de mutación

Podemos ver que el operador ‘sum’ ha generado dos mutantes y que ambas han muerto al pasar las pruebas sobre ellas, como consecuencia tenemos un 100% de porcentaje de mutación y el usuario sabe que sus pruebas están contemplando dicho caso. Lo mismo sucede para los mutantes generados con el operador ‘high’, solo que este ha generado un único mutante, pero no podemos decir lo mismo del operador ‘or’, el cual ha generado un mutante, pero este ha sobrevivido, por lo que implica que el porcentaje de mutación para las pruebas facilitadas en este caso es 0. Indicando al usuario que ningún caso generado por el cambio del operador or es detectado por las pruebas.

Sabiendo que solo un mutante ha fallado, el usuario puede recurrir a los resultados específicos en busca de que mutante concreto es el que ha sobrevivido, que mutación contiene y en que línea, para así poder realizar una nueva prueba o adaptar alguno de los existentes y así tener también este nuevo caso controlado.

En los resultados específicos se encontrará con lo que muestran las **Figuras 56,57,58,59,60**:

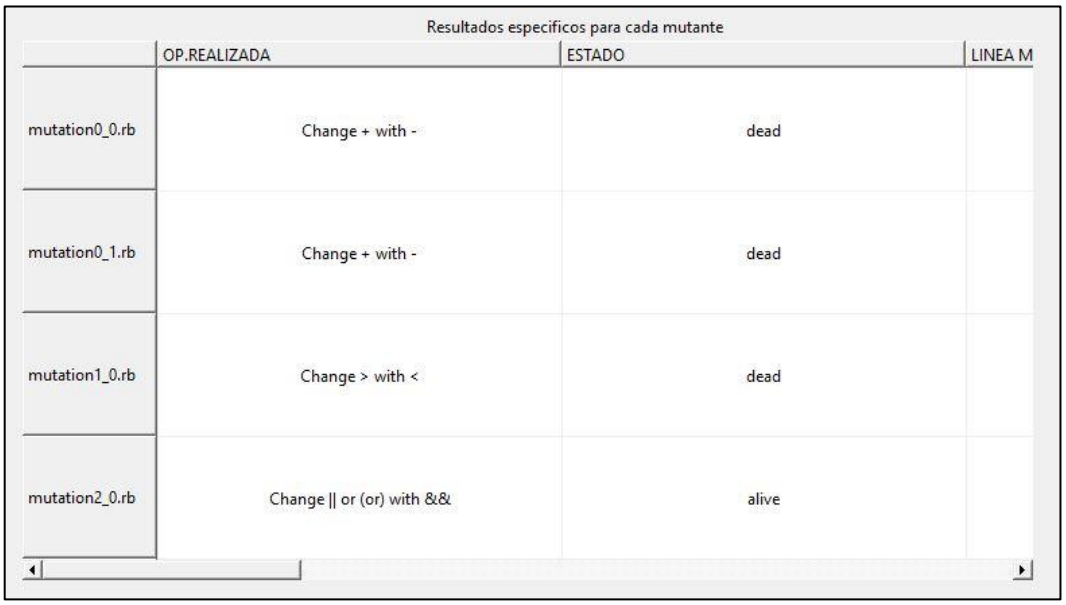


Figura 56 – Resultados por mutante

En la **Figura 56** que acabamos de ver, el usuario podrá ver que el mutante que ha sobrevivido ha sido el 2_0 y que ha sido por el reemplazo de una operación || (or) por una && (and).

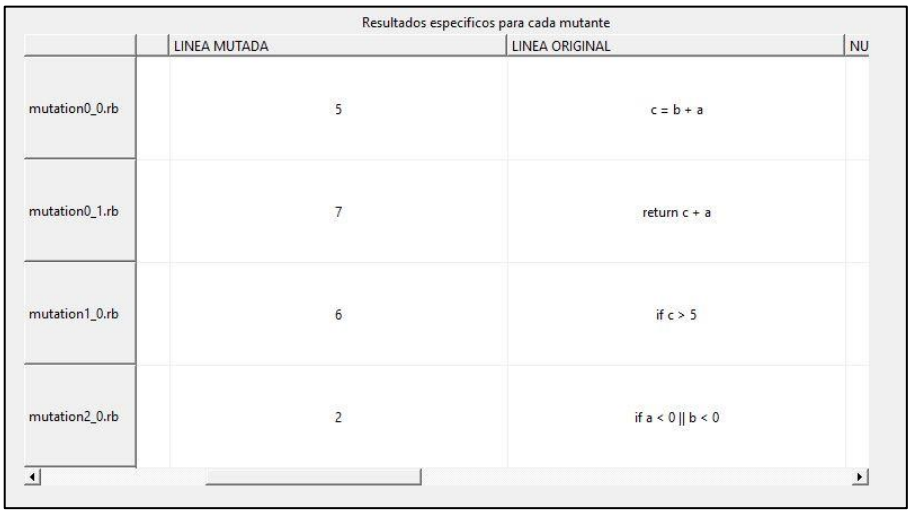


Figura 57 – Resultados por mutante

En la **Figura 57** que muestra la información que se puede obtener desde la misma tabla de la **Figura 56** desplazándonos con el scroll horizontal, el usuario puede observar que línea del fichero ha sido mutada y cual era el contenido original de dicha línea.

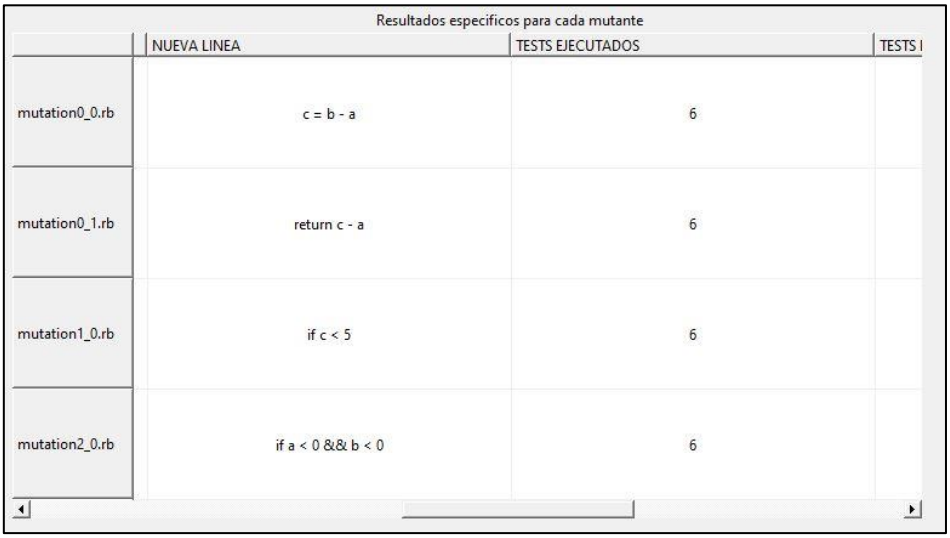


Figura 58 – Resultados por mutante

En la **Figura 58** el usuario puede ver la nueva línea que se ha producido tras realizar la mutación y esto le dará ya una mejor idea, junto con la línea original y el número de línea mutados ya vistos, de que test puede generar o cual de los que ya tiene no está siendo lo suficientemente robusto.

Finalmente podrá ver como datos adicionales los resultados obtenidos en las **Figuras 59 y 60**.

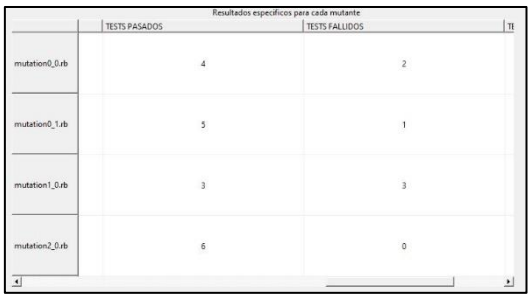


Figura 59 – Resultados por mutante

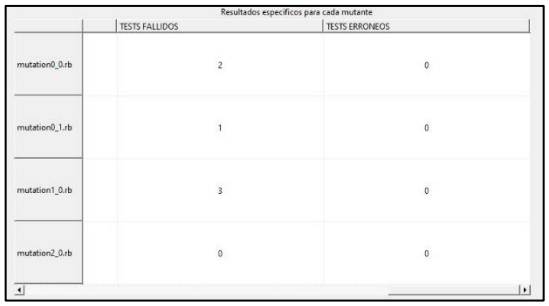


Figura 60 – Resultados por mutante

Si el usuario ha marcado la opción de guardar los mutantes generados, podrá abrir el contenido del mutante que quiera haciendo click sobre el nombre del fichero del mutante de esta tabla. En la **Figura 61**, se muestra un ejemplo de fichero mutation2_0.rb abierto. Esto permitirá al usuario visualizar la línea mutada dentro de la totalidad del código para ayudarlo en la búsqueda de la mejora de su conjunto de casos de prueba.

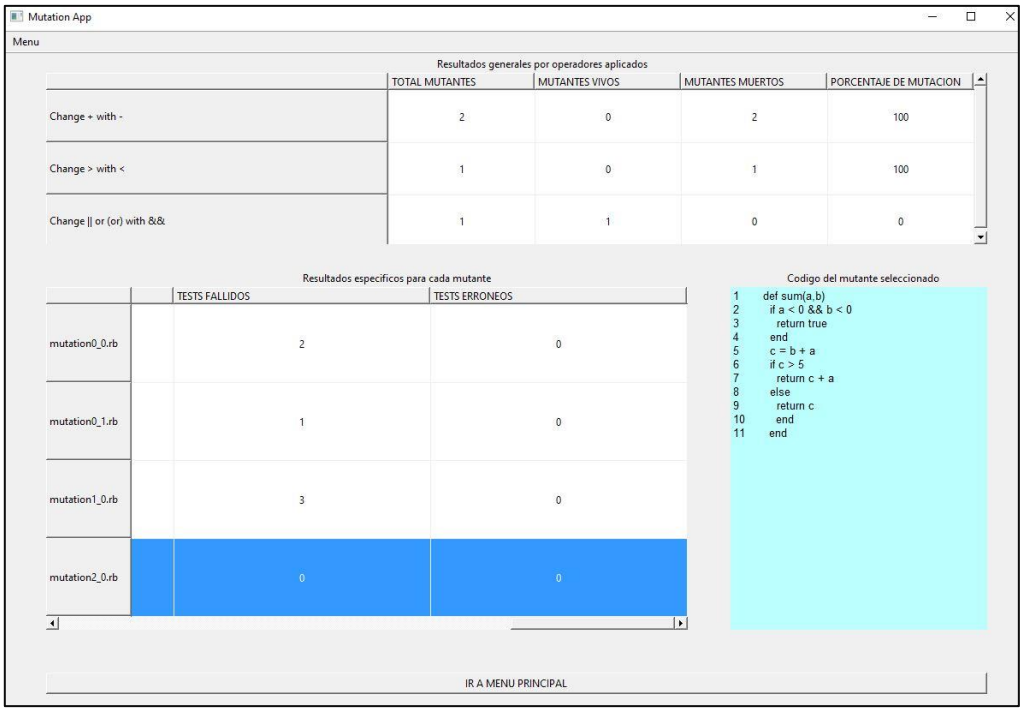


Figura 61 – Mostrado de contenido de mutantes en la aplicación

Como podemos observar en la **Figura 61**, el usuario puede visualizar las propias mutaciones dentro de la aplicación y comprender así mejor el mutante generado. Además, todos los resultados se muestran en la misma ventana, lo que permite al usuario poder consultar los distintos datos de forma conjunta sin necesidad de tener que estar navegando entre ventanas o vistas, los cual es un aspecto de gran utilidad para un desarrollador de código.

Como hemos podido ver en este anexo, los datos recopilados y mostrados en la aplicación, permiten al usuario tener una visión general y específica de las mutaciones realizadas. Además, los datos que proporciona permiten al usuario mejorar sus pruebas de casos de prueba para hacerlas más robustas y de mejor calidad. Permitiendo al usuario ampliar su batería de pruebas si así lo requiere o mejorar la calidad de las existentes.

En este caso concreto las pruebas test_return_true_expert y test_resturn_true de las **Figuras 52 y 53** respectivamente son los que no están detectando este cambio, así que deberán de mejorarse para que contemplen este caso o generar unas pruebas nuevas que si lo contemplen.

E Almacenamiento seguro de contraseñas

Cualquier aplicación software que almacene datos sensibles de los usuarios en sus bases de datos debe de contar con un sistema de seguridad eficaz que impida a un atacante obtener la información de estos en el caso de que sea capaz de robar dicha base de datos.

En nuestro caso, la aplicación puede contar, si el usuario la ha configurado, con una base de datos que almacenará datos sensibles como son las contraseñas de usuario, que estos pueden estar utilizando en otras aplicaciones. Por este motivo estos datos no pueden ser almacenados en texto plano y necesitan de un protocolo especial de tratamiento para ser almacenados.
Con este objetivo se ha optado por utilizar la estrategia de realizar un hash y un salt. Esta nos permitirá dotar de seguridad a nuestra base de datos en caso de que sea robada y el atacante intente extraer el contenido de las contraseñas mediante tablas pre-computadas o tablas arcoíris.

Estas tablas arcoíris son tablas de consulta que permiten obtener contraseñas de texto a partir del resultado de una función hash. Por tanto, para generarlas se realiza un script que comienza a crear todas las contraseñas posibles combinando todas las letras y caracteres que se pueda y realiza el hash de todas ellas. Una vez generada, va comprobando si alguno de los hashes obtenidos coincide con el que se encuentra en base de datos y en caso de ser así basta comprobar que contraseña de la tabla se corresponde con dicho hash para conocer la contraseña usada por un usuario.

Se muestra en la tabla siguiente un ejemplo de tabla precomputada donde la columna HASH contiene el valor Hash(x) que equivale al resultado de aplicar una función hash sobre la contraseña x.

CONTRASEÑA	HASH
a	Hash(a)
b	Hash(b)
. . .	
abc	Hash(abc)
abd	Hash(abd)
. . .	
fr_astse	Hash(fr_astse)
fr_astsf	Hash(fr_astsf)
fr_astsg	Hash(fr_astsg)
. . .	

Tabla 2 - Tabla Arcoíris de ejemplo

Este tipo de tablas es capaz de revertir contraseñas de hasta 8 caracteres haseadas con el hash MD5 de forma inmediata. Por lo que defender nuestra base de datos utilizando únicamente el hash de la contraseña no es una buena opción.

Si observamos bien este tipo de ataques tiene dos aspectos que son fundamentales en su funcionamiento, el tiempo y el espacio. Cuanto más largas sean las contraseñas en la base de datos más tiempo necesitará para calcularlas todas y además más combinaciones distintas habrá y por tanto más espacio en disco necesitará para guardarlas todas.

Por lo tanto, si concatenamos una cadena aleatoria (salt) a la contraseña antes de calcular su hash, provocará que el tiempo y espacio que necesiten estas tablas sea tan alto que no sea un método asequible de llevar acabo y además provocamos que el hash asociado a una clave sea único, de tal modo que dos claves iguales tengan distinto hash y si una de ellas es descifrada por algún motivo, la otra sigue estando protegida.

El proceso de generación por tanto sería:

1 – Se calcula un salt aleatorio DIFERENTE PARA CADA CONTRASEÑA, que no tiene que ser secreto obligatoriamente, pero si impredecible.
2 – Se hasea la contraseña y el salt juntos.
3 – Se almacena en base de datos el resultado del hash y el salt junto con el resto de la información del usuario.

Y por tanto cuando un usuario acceda a la aplicación mediante su usuario y contraseña, se verificará que la contraseña es correcta del siguiente modo:

1 – Se obtiene el salt del usuario que está accediendo.
2 – Se calcula el hash con la contraseña que ha introducido el usuario al conectarse.
3 – Se comprueba si el resultado del hash realizado es el mismo que hay en base de datos.

Además, el salt utilizado debe ser lo suficientemente largo, ya que un salt demasiado corto no soluciona el problema.

En el caso de esta aplicación el tamaño del salt generado es de 30 caracteres.

El siguiente es un ejemplo de salt generado que se puede ver en la **Figura 15**:

$2a$12$mxRQ08U66Cz1UzvnpmlKDe

Y la función hash utilizada ha sido bcrypt que está basada en el cifrado de Blowfish[10].

Para poder usarla se ha utilizado la gema de Ruby que tiene el mismo nombre bcrypt que permite utilizar esta función y se encarga de autogestionar la creación de salts aleatorios [11].

En la **Figura 62**, se puede observar el código implementado de la aplicación desarrollada para gestionar el tratamiento de contraseñas al crear un usuario haciendo uso de la gema bcrypt.

```ruby
def create_user(name, password)
  # create a user in the database
  # :param name: user name
  # :param password: user password
  # :return: true if the user was created and false if one with that name already existed, -1 if the table
  # not exist
  begin
    hash, salt = save_passwords(password)
    @@conn.exec "INSERT INTO users VALUES('#{name}','#{hash}', '#{salt}')"
  rescue PG::Error, PG::UndefinedTable => e
    if e.instance_of?(PG::UndefinedTable)
      return -1
    else
      return false
    end
  end
  return true
end

def save_passwords(password)
  # Given a password, generate a random salt and concatenate this salt with the password
  # to generate a hash
  # :param password: user password
  # :return: a the password hashed with the salt and the salt
  salt = BCrypt::Engine.generate_salt # tam 29
  hash = BCrypt::Engine.hash_secret(password, salt) # tam 30
  return hash, salt
end
```

Figura 62 – Codificación del registro de la aplicación

Y en la **Figura 63**, se puede observar el código desarrollado para comprobar la autenticación de usuarios.

```ruby
def login(name, password)
  # Given a name and a password check that the name exists and the password is correct
  # :param name: user name
  # :param password: user password
  # :return: true if the authentication is correct, false if the name or password is incorrect, -1 if the table
  # not exist
  begin
    rs = @@conn.exec "SELECT * FROM users WHERE name = '#{name}'"
  rescue PG::UndefinedTable
    return -1
  end

  begin
    user_info = rs.collect.next # Get first row
  rescue Exception => e # if the query has not returned elements, the name entered is wrong
    return false
  end

  password_in_db = user_info['password']
  salt = user_info['salt']
  hash = BCrypt::Engine.hash_secret(password, salt)

  if password_in_db == hash
    return true
  else
    return false
  end

end
```

Figura 63 – Codificación de login de la aplicación

F Requisitos de la aplicación

En este apartado se mostrarán los requisitos tanto funcionales como no funcionales que se han considerado importantes para el diseño de la aplicación. Donde funcionales son aquellos que agregan funcionalidad y no funcionales aquellos importantes para la calidad y usabilidad.

Requisitos funcionales:

RF.1 – Registro de usuarios: permitir registrar usuarios mediante un nombre que aún no exista en el sistema y una contraseña en caso de estar la base de datos configurada.
RF.2 – Acceso con usuario: permitir acceder al sistema mediante un nombre y contraseña que hayan sido registrados con anterioridad y cuando la base de datos esté configurada.
RF.3 – Acceso sin usuario: permitir al usuario acceder sin necesidad de usar usuario y contraseña sin importar si la base de datos está configurada o no.
RF.4 – Cualquier tipo de usuario: poder seleccionar el fichero que se desea mutar.
RF.5 – Cualquier tipo de usuario: poder seleccionar las pruebas que se desean aplicar tras una mutación.
RF.6 – Cualquier tipo de usuario: poder seleccionar de forma múltiple los operadores que se desean utilizar para generar las mutaciones.
RF.7 – Cualquier tipo de usuario: seleccionar si desea guardar las mutaciones que se generen o no.
RF.8 – Cualquier tipo de usuario: visualizar los resultados de cada mutación generada por los operadores seleccionados, para el fichero escogido, tras pasar las pruebas deseadas.
RF.9 – Cualquier tipo de usuario: poder visualizar el contenido de las mutaciones obtenidas dentro de la aplicación en caso de haber seleccionado el salvado de estas.
RF.10 – Un usuario registrado: poder crear operadores con un nombre que no exista previamente en la aplicación ya por defecto o que ya haya sido creado por el mismo usuario.
RF.11 – Un usuario registrado: eliminar operadores que el haya creado.
RF.12 – Un usuario registrado: poder seleccionar operadores creados por él mismo para aplicar en una mutación.
RF.13 – Un usuario sin registro: poder crear operadores con un nombre que no exista previamente en la aplicación por defecto o que ya haya sido creado anteriormente por otro usuario que accediera sin registro.
RF.14 – Un usuario sin registro: poder eliminar operadores creados por él o por otros usuarios que hayan accedido con el modo sin registro.
RF.15 – Un usuario sin registro: poder seleccionar operadores creados por él mismo o por otro usuario que haya accedido con el modo sin registro para aplicar en una mutación.
RF.17 – Cargar operadores de mutación: cargar los operadores por defecto que vienen con la aplicación en un .csv en el sistema.
RF.18 – Generar mutaciones: a partir del AST generar las mutaciones al código especificado aplicando los operadores de mutación seleccionados por el usuario.
RF.19 – Aplicar pruebas unitarias: aplicar las pruebas unitarias seleccionadas por el usuario a las mutaciones generadas.

RF.20 – Gestionar resultados y mostrar correctamente: gestionar los resultados obtenidos de pasar las pruebas a las mutaciones y organizarlos para mostrarlos ordenadamente de forma clasificada.
RF.21 – Gestionar base de datos: en caso de existir la base de datos para gestionar los usuarios, la aplicación debe de ser capaz de conectarse a ella y poder gestionarla mediante consultas de forma correcta para poder generar nuevas entradas con nuevos usuarios y comprobar si un usuario existe con nombre y contraseña.
RF.22 – Gestionar errores: gestionar los errores adecuadamente y realizando en cada ocasión la operación que proceda (mostrar mensaje, evitar acceso a fichero no existente …).

Requisitos no funcionales:

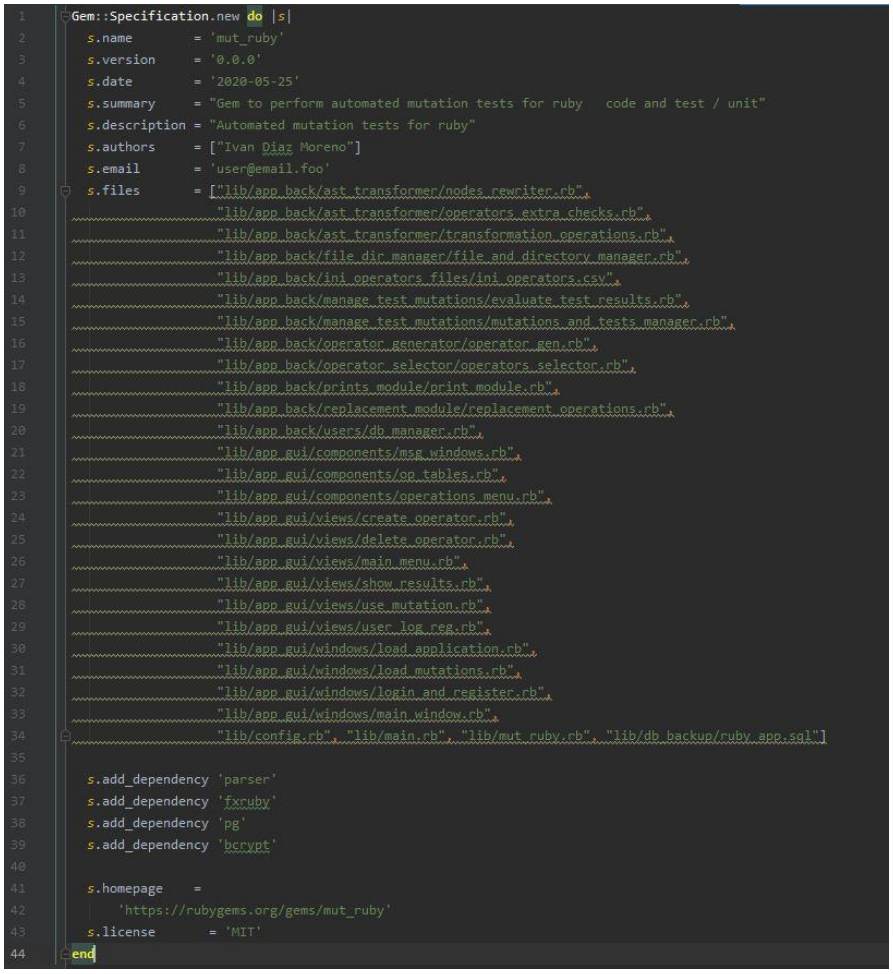
RNOF.1 – La aplicación será instalable en entornos locales mediante una gema.
RNOF.2 – La aplicación se podrá ejecutar sin conexión a internet.
RNOF.3 – La aplicación se podrá ejecutar en dos modos: con gestión de usuarios si la base de datos está configurada en el sistema o sin gestión de usuarios tanto si la base de datos está configurada como si no.
RNOF.4 – La aplicación contará con ventanas de aviso para mantener al usuario informado de todos los casos que puedan ir ocurriendo.
RNOF.5 – El uso del modo con gestión de usuarios requerirá la configuración de una base de datos PostgreSQL de la forma que se indica en la documentación.
RNOF.6 – El proyecto se encontrará alojado en un repositorio de GIT con un readme.md que facilite al usuario la instalación y configuración de la aplicación.
RNOF.7 – En caso de usarse el modo con gestión de usuarios, las contraseñas serán guardadas en base de datos utilizando la técnica de hash + salt.
RNOF.8 – El conjunto de los operadores por defecto de la aplicación deben de ser fácilmente ampliables.
RNOF.9 – El sistema debe hacer uso del AST para realizar el análisis del código y sus modificaciones.

G Generación de una gema

En este Anexo se muestra el proceso seguido para hacer de la aplicación desarrollada una gema y que de este modo su instalación sea lo más sencilla posible.

Para generar la gema mut_ruby que permite instalar la aplicación de forma sencilla, se ha tenido que añadir un fichero llamado mut_ruby.gemspec cuyo contenido se puede observar en la **Figura 64**.

Este fichero contiene toda la información necesaria para poder generar una gema, contiene el nombre que va a tener, la fecha de creación, la versión, una descripción, etc. Pero entre los campos más importantes se encuentran el campo *files*, en el cual hay que especificar todos los ficheros que contiene el programa del cual se quiere generar la gema y la función *add_dependecy* que permite agregar dependencias entre gemas. De este modo se especifican con *add_dependecy* todas aquellas gemas de las que depende nuestro programa para que estas sean instaladas de forma automática al realizar la instalación de nuestra gema si no están ya instaladas. A estas se las puede especificar una versión concreta si nuestra aplicación requiere de alguna versión específica para su correcto funcionamiento. En nuestro caso no hay una dependencia de versión con ninguna de ellas por lo que al no especificarla, se instalará la versión más actualizada para cada una.



```
Gem::Specification.new do |s|
  s.name        = 'mut_ruby'
  s.version     = '0.0.0'
  s.date        = '2020-05-25'
  s.summary     = "Gem to perform automated mutation tests for ruby   code and test / unit"
  s.description = "Automated mutation tests for ruby"
  s.authors     = ["Ivan Diaz Moreno"]
  s.email       = 'user@email.foo'
  s.files       = ["lib/app_back/ast_transformer/nodes_rewriter.rb",
                   "lib/app_back/ast_transformer/operators_extra_checks.rb",
                   "lib/app_back/ast_transformer/transformation_operations.rb",
                   "lib/app_back/file_dir_manager/file_and_directory_manager.rb",
                   "lib/app_back/ini_operators_files/ini_operators.csv",
                   "lib/app_back/manage_test_mutations/evaluate_test_results.rb",
                   "lib/app_back/manage_test_mutations/mutations_and_tests_manager.rb",
                   "lib/app_back/operator_generator/operator_gen.rb",
                   "lib/app_back/operator_selector/operators_selector.rb",
                   "lib/app_back/prints_module/print_module.rb",
                   "lib/app_back/replacement_module/replacement_operations.rb",
                   "lib/app_back/users/db_manager.rb",
                   "lib/app_gui/components/msg_windows.rb",
                   "lib/app_gui/components/op_tables.rb",
                   "lib/app_gui/components/operations_menu.rb",
                   "lib/app_gui/views/create_operator.rb",
                   "lib/app_gui/views/delete_operator.rb",
                   "lib/app_gui/views/main_menu.rb",
                   "lib/app_gui/views/show_results.rb",
                   "lib/app_gui/views/use_mutation.rb",
                   "lib/app_gui/views/user_log_reg.rb",
                   "lib/app_gui/windows/load_application.rb",
                   "lib/app_gui/windows/load_mutations.rb",
                   "lib/app_gui/windows/login_and_register.rb",
                   "lib/app_gui/windows/main_window.rb",
                   "lib/config.rb", "lib/main.rb", "lib/mut_ruby.rb", "lib/db_backup/ruby_app.sql"]

  s.add_dependency 'parser'
  s.add_dependency 'fxruby'
  s.add_dependency 'pg'
  s.add_dependency 'bcrypt'

  s.homepage    =
     'https://rubygems.org/gems/mut_ruby'
  s.license       = 'MIT'
end
```

Figura 64 – Contenido fichero mut_ruby.gemspec

Una vez que este fichero ha sido generado, hay que asegurarse que el fichero de arranque de nuestra aplicación conocido en muchos lenguajes como main, debe de tener el mismo nombre que la gema, por lo que, para el caso de este proyecto, el nombre del main será mut_ruby.rb.

Con el fichero .gemspec en el directorio del proyecto y el fichero de arranque con el nombre de la gema ya puesto, se puede proceder a realizar el comando:

```
gem build mut_ruby.gemspec
```

Una vez que es ejecutado el comando, en el mismo directorio del proyecto se generará la gema correspondiente mut_ruby-x.y.z.gem, donde x,y,z son el número de versión especificado en el fichero .gemspec

Para instalar la aplicación y todas las gemas de las que depende ahora bastará con realizar el comando:

```
gem install mut_ruby-x.y.z.gem
```

H Recorrido árbol post-orden.

La estructura de programación árbol, es un tipo abstracto de datos, usado comúnmente en programación, que permite almacenar información y recorrerla de una forma sencilla. Los árboles cuentan con dos elementos básicos que los componen. Por un lado, están los nodos y por otro las ramas. Cada nodo se puede conectar a uno a más nodos mediante las ramas. De este modo el nodo superior del árbol es denominado nodo raíz o padre y todos los nodos a los que se pueden llegar desde él mediante una rama, son denominados sus hijos. Finalmente, aquellos nodos que no tienen hijos son denominados nodos hoja.

En la **Figura 65** se puede observar un ejemplo de árbol no binario, es decir, un árbol en el que cada nodo puede tener más de dos hijos.

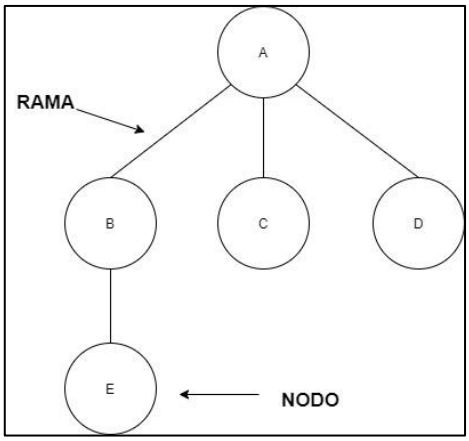


Figura 65 - Árbol ejemplo

Cada árbol contiene subárboles, por ejemplo los nodos B y E, conectado mediante una rama, de la **Figura 68** forman un subárbol.

Los árboles pueden ser recorridos en profundidad o en anchura. Esto permite visitar cada nodo el árbol, si exceptuar ninguno, exactamente una vez. Dependiendo del modo en que sean recorridos, se varía el orden en el que se visitan los nodos.

En el caso de este proyecto, para recorrer los AST que se generan a partir del código de entrada, se ha decidió recorrer el AST en profundidad. Dentro de este modo hay distintas variantes, pero en nuestro caso hemos hecho caso del modo post-orden. Este consiste en realizar los siguientes pasos:

1- Recorrer primero el subárbol izquierdo.
2- Recorrer segundo el subárbol derecho.
3- Visitar el nodo raíz.

Veamos un ejemplo para que quede un poco más claro. Supongamos el árbol de la **Figura 69**[9].

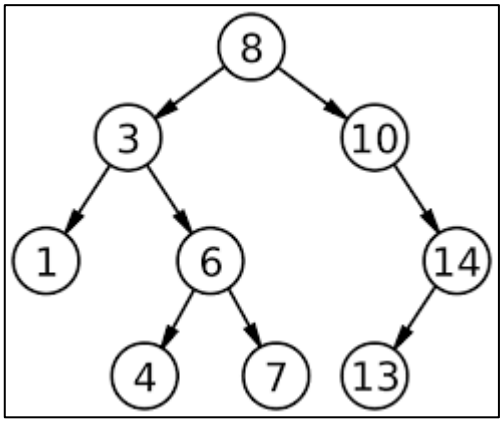


Figura 66 - Árbol de búsqueda.

El orden en el que se visitarían los nodos de este árbol de la **Figura 66** en profundidad post-orden, sería:

1 – 4 – 7 – 6 – 3 – 13 – 14 – 10 – 8

Como podemos observar primero se recorre el subárbol izquierdo de cada nodo, después el subárbol derecho y finalmente en nodo raíz de cada subárbol. De este modo se recorren todos los nodos exactamente una vez y sin excluir ninguno de ellos.